UC Davis
UC Davis Electronic Theses and Dissertations

Title

AI-Driven ASIC Design and Verification: Integrating Machine Learning, LLMs, and Secure Hardware Practices

Permalink

https://escholarship.org/uc/item/9dk022mm

ISBN

9798297645196

Author

Jin, Yuyang

Publication Date

2025-09-21

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

AI-Driven ASIC Design and Verification: Integrating Machine Learning, LLMs, and Secure Hardware Practices

by

YUYANG JIN

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

Davis

Approved:

Houman Homayoun, Chair

Bevan Baas

Hussain Al-Asaad

Committee in Charge

2025

AI-Driven ASIC Design and Verification: Integrating Machine Learning, LLMs, and Secure Hardware Practices

Copyright 2025
by
Yuyang Jin

Abstract

AI-Driven ASIC Design and Verification: Integrating Machine Learning, LLMs, and Secure Hardware Practices

by

Yuyang Jin

MASTER OF SCIENCE in Electrical and Computer Engineering

University of California, Davis

Houman Homayoun, Chair

The growing complexity of Application-Specific Integrated Circuits (ASICs) has driven the demand for more efficient design and verification methodologies. This thesis explores a range of techniques and toolchains that span the full ASIC flow from register-transfer level (RTL) design and verification to layout generation and security evaluation.

We begin with the design of functional RTL components such as finite-state machines (FSMs), matrix multipliers, an Advanced High-Performance Bus (AHB) to SRAM controller, and a small neural network accelerator. We establish functional correctness through simulation (Synopsys VCS with Verdi), functional and code coverage, and selected formal checks. For physical design, we demonstrate RTL-to-GDSII using OpenLane with the Sky130 PDK and perform logic synthesis and timing analysis via Synopsys Design Compiler, reporting configuration choices and the resulting area and timing for our designs.

We then review applications of machine learning in hardware design and verification, with emphasis on large language model (LLM)-assisted methods. In our work, LLMs are applied to module-level verification support, including automatic generation of testbenches and checkers.

Finally, we present real-world case studies in hardware security, centered on vulnerabilities in the OpenTitan SoC platform. Using formal tools (VC Formal, SpyGlass), taint tracking, and custom bug-detection scripts, we uncover security flaws and propose automated mitigation workflows.

Together, these contributions demonstrate a comprehensive methodology for secure, verifiable ASIC design using both traditional and AI-augmented toolchains.

Contents

Contents iv

List of Figures vi

List of Tables vii

1 Introduction 1
- 1.1 Challenges in Modern Hardware Design . . . 1
- 1.2 Rise of Machine Learning in EDA . . . 2
- 1.3 From Verification Bottlenecks to LLM Automation . . . 2
- 1.4 Thesis Scope and Contributions . . . 3

2 Digital IC Design 4
- 2.1 Brief Background . . . 4
- 2.2 FSM Design . . . 5
- 2.3 Matrix Multiplication Design . . . 7
- 2.4 AlexNet Convolution Accelerator Design . . . 9
- 2.5 AHB-SRAM Controller Design . . . 15
- 2.6 Summary . . . 20

3 Hardware Verification with Synopsys VCS 21
- 3.1 VCS at a Glance . . . 21
- 3.2 Environment Setup . . . 22
- 3.3 Compilation and Simulation . . . 23
- 3.4 Interactive Debug with Verdi . . . 23
- 3.5 Coverage Analysis . . . 24
- 3.6 Runtime Profiling . . . 24
- 3.7 Workflow Checklist . . . 25
- 3.8 Build Automation Scripts . . . 26

4 ASIC Physical Design and EDA Toolchains 30
- 4.1 RTL-to-GDSII: Conceptual Overview . . . 30
- 4.2 OpenLane Flow . . . 31

4.3 Case Study 1: picorv32 RISC-V Core 31
4.4 Case Study 2: Scratchpad Memory (SPM) 34
4.5 RTL Synthesis with Design Compiler 34
4.6 Discussion and Summary 37

5 Machine Learning for Hardware 38
5.1 Related Work: Pyramid Framework 38
5.2 Case Study: ML-Guided Timing Deviation Prediction 40
5.3 Summary 42

6 LLM-Assisted ASIC Design 43
6.1 Emerging Roles of LLMs in ASIC Design 43
6.2 Representative Case Studies 44
6.3 Discussion and Outlook 46

7 Hardware Security 48
7.1 Overview: Security Verification in Practice 48
7.2 Hack@DAC: Competition Background 48
7.3 Toolchain Overview: Synopsys + Custom Automation 49
7.4 Automated Detection Workflow 52
7.5 Case Studies: Three Exploit Examples 52
7.6 Summary and Reflections 54

8 Conclusion and Future Work 55

Bibliography 56

List of Figures

3.1 Why VCS is central to the digital-IC design flow 22
3.2 Verdi nWave session showing the four-state FSM waveform [15] 24
3.3 Code-coverage dashboard generated by URG [15] 25
3.4 Simulation time breakdown using `simprofile` and `profrpt` [15] 26

4.1 GDSII layout view of `picorv32` synthesized via OpenLane. 33
4.2 GDSII layout of `spm` memory block post-OpenLane flow. 35
4.3 Timing report from Design Compiler showing critical path delay matching the 6 ns target [18]. 36
4.4 Area breakdown report from Design Compiler for `alex35_final` [18]. 36

5.1 Representative ML applications in HLS design tasks, adapted from [4]. 39
5.2 Reconstructed flow of Pyramid: HLS feature extraction and ML-driven design prediction [6]. 40
5.3 SHAP Analysis for Feature Importance 41

7.1 VCS and Verdi used to simulate OpenTitan IPs and trace internal FSM signals [15]. 50
7.2 VC Formal used for assertion coverage and sequential equivalence checking [35]. 51
7.3 SpyGlass Lint output used to catch coding flaws and reset misbehavior [36]. . . 51
7.4 TPROP-assisted assertion failure highlighting propagation flaw in `alert_tx_o` output [37]. 52
7.5 Custom LLM and script-based workflow for automated test generation. 53

List of Tables

3.1 Minimal two-step VCS flow for RTL verification. 26

6.1 Representative LLM-Based Frameworks for ASIC Design and Verification 44

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my advisor, Professor Houman Homayoun, for his unwavering support, insightful guidance, and continuous encouragement throughout every stage of this thesis. His mentorship has been instrumental in shaping both my technical growth and academic direction.

In addition, I am especially thankful to my mentor, Kevin Immanuel Gubbi, whose patience, dedication, and inspiring mentorship have made a significant impact on my research experience. Working alongside him has been both intellectually enriching and personally fulfilling. His encouragement not only deepened my understanding of the subject matter but also sparked a lasting interest in the field. Moreover, I feel fortunate to have built a meaningful friendship through our collaboration.

I would also like to extend my sincere appreciation to Professors Bevan Baas and Hussain Al-Asaad for kindly serving on my thesis committee. Their thoughtful feedback and valuable insights have greatly contributed to the quality and clarity of this work.

Furthermore, I am grateful to my colleagues and peers in the ASEEC Lab for their stimulating discussions, generous support, and collaborative spirit. Their contributions helped foster a motivating and productive research environment.

Finally, and most importantly, I wish to thank my family for their unwavering love, patience, and belief in me. Their constant support has provided the foundation upon which all of my accomplishments are built.

Chapter 1

Introduction

Over the past few decades, integrated circuit (IC) design has undergone a dramatic transformation. From the era of handcrafted layouts and logic minimization charts to today's vast systems-on-chip (SoCs) hosting billions of transistors, the evolution has been both deep and fast. This shift has been driven in part by emerging application domains such as AI accelerators, autonomous driving systems, and high-speed networking, all of which demand increasingly sophisticated hardware.

Back in the 1980s and 90s, the focus of chip design was different. Designers worked with limited transistor budgets, and tools were basic compared to today's standards. Techniques like schematic capture and gate-level simulation were the norm. But things changed rapidly. The introduction of hardware description languages such as Verilog and VHDL opened the door to RTL-based design, making systems easier to describe and scale.

The design flow, from RTL to layout, has grown more complex, requiring toolchains such as Synopsys Design Compiler, Verdi, and open-source frameworks like OpenLane [1]. As the complexity of modern SoCs increases—with multiple cores, dedicated accelerators, and security features packed into a single chip—designers face a real struggle to explore the massive space of possible implementations and verify their correctness. The challenges today go beyond just fitting everything on a chip. Power, area, timing, security, and design time all compete for attention. Manual engineering, once enough for simple processors or controllers, now falls short. We need smarter tools, better abstractions, and more automation to keep up. That's the context in which this thesis was written: a landscape where understanding the full hardware stack, from RTL through verification and down to layout, is more crucial than ever.

1.1 Challenges in Modern Hardware Design

Modern design challenges are multifaceted. From a performance standpoint, computing systems must deliver both throughput and energy efficiency. Borkar and Chien [2] outlined how increasing transistor budgets no longer translate directly into performance due to power

and reliability limits. They argued for a new direction: specialization and smarter design automation. At the same time, software and hardware development lifecycles remain disconnected. Bridging this gap demands automation across all stages—design, verification, testing, and optimization.

One promising response to these challenges has been the use of High-Level Synthesis (HLS) tools, which allow designers to describe hardware in C/C++ rather than Verilog/VHDL. Cong et al. provided a comprehensive analysis of the state-of-the-art in HLS, highlighting recent advances and persistent gaps in productivity and design quality [3]. They emphasized that while HLS shows great promise in abstracting hardware implementation, tool usability and predictability of resource and timing estimates remain core obstacles. The adoption of machine learning (ML) to predict these outcomes has been gaining momentum.

1.2 Rise of Machine Learning in EDA

The application of machine learning techniques to Electronic Design Automation (EDA) has grown rapidly over the past five years. Huang et al. [4] surveyed a wide range of ML applications across the EDA stack—from floorplanning and placement to routing, timing, and logic optimization. Their work categorized models into supervised, unsupervised, and reinforcement learning, illustrating how learning-based tools outperform hand-tuned heuristics in many EDA sub-tasks.

A complementary survey by Cong et al. [5] presented detailed insights on integrating ML into physical design and circuit optimization. The consensus is that ML can assist with fast design space exploration, prediction of tool outcomes (such as delay and power), and even with guiding synthesis and verification. One representative framework is Pyramid [6], which used ML models to predict timing and resource usage of HLS designs. It combines code features and synthesis attributes to train regression models that reduce redundant synthesis runs. Similarly, HLSdataset [7] offers an open-source dataset containing HLS design examples annotated with resource and timing labels, enabling further ML research in this space.

In our study, we extended this line of work by incorporating interpretability techniques from explainable AI. Specifically, we used SHAP (SHapley Additive exPlanations) [8] to attribute the influence of design features, such as loop depth, number of multiplications, or memory usage—on downstream outcomes like latency or area. This analysis revealed that arithmetic intensity is a key factor influencing HLS scheduling delays. Such insights improve trust and transparency when integrating ML into hardware flows.

1.3 From Verification Bottlenecks to LLM Automation

Verification continues to be the dominant cost and time bottleneck in the hardware development cycle. Functional simulation, formal property checking, and assertion-based verification

require significant manual effort. Inspired by the success of large language models (LLMs) in software engineering, researchers have recently explored how these models can assist with testbench generation, assertion inference, and bug localization.

Qiu et al. introduced AutoBench [9], a pioneering framework that uses LLMs to automatically generate and evaluate HDL testbenches. It demonstrated that models like ChatGPT can learn test patterns and interface protocols from design context alone. Building on this, CorrectBench [10] added a self-correction mechanism that iteratively improves testbench quality via feedback loops. The broader research landscape now includes tools like LLM4DV [11], which generates test stimuli, and HDLCoRe [12], which mitigates hallucinations in generated code. The potential of LLMs to accelerate verification is clear, though practical deployment remains dependent on improving the robustness, context understanding, and integration of such models into industrial flows.

1.4 Thesis Scope and Contributions

This thesis presents a comprehensive journey through the design, simulation, and verification of digital systems, integrating both classical methods and recent advances in ML and LLM-driven automation. Key contributions include:

- A hands-on demonstration of RTL simulation, debugging, and coverage analysis using Synopsys VCS and Verdi [13].
- ASIC physical design using OpenLane, with practical walkthroughs of RTL-to-GDSII flow on open-source RISC-V and custom accelerator designs [1].
- A survey and implementation case studies of ML-guided estimation tools, including Pyramid [6] and AutoBench [9].
- A critical assessment of LLM-based verification generation tools, with testbench results, functional coverage metrics, and proposed enhancements.

Through this work, we aim to bridge the gap between conventional digital IC education and emerging AI-assisted design methodologies, equipping future engineers with both foundational and forward-looking skills.

Chapter 2

Digital IC Design

2.1 Brief Background

Digital integrated-circuit (IC) design lies at the heart of modern computing, enabling the transformation of behavioral intent into concrete hardware realizations. Over the course of this thesis, I progressed from designing simple control logic to more sophisticated datapath architectures and memory-centric accelerators. This chapter introduces four RTL designs that serve as foundational case studies: a four-state finite-state machine (FSM), a parameterized matrix-multiplication engine, a CNN-style convolution accelerator, and an AHB-compliant SRAM controller. Each was implemented in synthesizable Verilog and verified through simulation using Synopsys VCS and waveform inspection via Verdi.

The FSM design, while modest in complexity, captures the full RTL design loop. It models a control structure with a clean separation between sequential state storage, combinational next-state logic, and output generation. This three-block architecture emphasizes synchronous design discipline and is commonly found in bus protocols, configuration sequencers, and peripheral interfaces. It also served as an ideal starting point for waveform debugging and coverage collection, laying the groundwork for the simulation methodology in Chapter 3.

The matrix multiplication engine shifts the focus to compute-intensive datapath design. As a core computational kernel in digital signal processing and machine learning, efficient matrix multiplication requires attention to nested iteration, signed arithmetic, and data structure flattening. The RTL module is parameterized by matrix dimension (`N`) and operand bit-width (`width`), enabling flexible scaling from testbench prototypes to SoC accelerator blocks. Although currently implemented as a combinational unit, it provides a platform for further pipelining and resource sharing.

The third design is a CNN-inspired convolution accelerator, modeled on the first layer of AlexNet [14]. The design performs tiled 11×11 convolution on 35×35 image patches, applies ReLU activation, and conducts 2×2 max pooling. Internally, it uses an FSM to orchestrate memory loading, convolution, and post-processing. While the implementation is

time-stepped rather than deeply pipelined, the design exposes all internal buffers and data paths, making it a useful case study for exploring cycle budgeting, parallelism, and buffer reuse.

The final design, an AHB-SRAM controller, demonstrates bus protocol compliance and memory system integration. Built for the AMBA AHB-Lite bus, it supports IDLE, READ, WRITE, and BIST modes. The controller handles all AHB protocol signals and manages address decoding, chip select generation, and pipelined handshaking for read/write operations. It bridges the gap between high-level bus transactions and low-level SRAM access, and provides a robust platform for backend synthesis and layout exploration in Chapter 4.

2.2 FSM Design

A four-state finite-state machine (FSM) was implemented in Moore style, with distinct states labeled `S0` through `S3`. The complete SystemVerilog RTL is shown in Listing 2.1. The FSM transitions sequentially from `S0` to `S3` and then wraps around, creating a simple yet complete example of control-dominated digital logic. A minimal testbench applies a 50% duty-cycle clock, asserts asynchronous reset, and runs the circuit for 20 cycles, verifying the expected repeating pattern:

$$S0 \rightarrow S1 \rightarrow S2 \rightarrow S3 \rightarrow S0 \rightarrow \ldots$$

Listing 2.1: Full RTL listing for the four-state FSM

```
module fsm (
    input C,
    input clk,
    input rst_n,
    output wire Y
);

parameter S0 = 2'b00;
parameter S1 = 2'b01;
parameter S2 = 2'b10;
parameter S3 = 2'b11;

reg [1:0] current_state, next_state;
reg reg_Y;

assign Y = reg_Y;

// State register
always @(posedge clk or negedge rst_n) begin
    if (!rst_n)
        current_state <= S0;
    else
        current_state <= next_state;
```

```
24 end
25
26 // Next state logic
27 always @(*) begin
28     case (current_state)
29         S0: next_state = C ? S1 : S0;
30         S1: next_state = C ? S1 : S2;
31         S2: next_state = C ? S3 : S2;
32         S3: next_state = C ? S3 : S0;
33         default: next_state = S0;
34     endcase
35 end
36
37 // Output logic
38 always @(*) begin
39     case (current_state)
40         S0: reg_Y = 0;
41         S1: reg_Y = 0;
42         S2: reg_Y = 1;
43         S3: reg_Y = C ? 1 : 0;
44         default: reg_Y = 0;
45     endcase
46 end
47
48 endmodule
```

Implementation Notes

- **State Encoding**: A compact 2-bit binary encoding was used to represent the four states. The RTL is agnostic to encoding and could easily support one-hot or Gray coding if needed.

- **Reset Strategy**: The FSM includes an asynchronous, active-low reset to ensure that the machine reliably enters `S0` at power-up or upon external reset assertion. This choice simplifies verification and aligns with common ASIC reset conventions.

- **Design Structure**: The RTL uses the classic three-block FSM structure: one block for the state register, one for next-state logic, and one for output logic. This separation clarifies timing paths and enables straightforward coverage-driven simulation.

- **Verification Readiness**: The FSM was developed primarily to support waveform-based debug and coverage analysis. Although not synthesized, it exercised key RTL coding patterns (e.g., `case` statements, parameterized state labels, and Moore output logic) and served as a reference point for setting up the Synopsys VCS flow in Chapter 3.

2.3 Matrix Multiplication Design

Matrix multiplication is a foundational operation in many computing domains, ranging from scientific computing to modern machine learning and signal processing. At the heart of this operation lies the multiply-accumulate (MAC) unit, which performs repeated multiplication and summation—an ideal candidate for hardware acceleration due to its regular structure and high arithmetic intensity. Efficient implementation of MAC operations is critical for achieving high performance in digital signal processors (DSPs), graphics processing units (GPUs), and neural network accelerators.

Listing 2.2 shows the complete RTL for a parameterizable matrix-multiplication accelerator. The design streams in two $N \times N$ unsigned 16-bit matrices and computes a 32-bit result matrix. It uses a three-level nested-loop controller to issue multiply-accumulate operations in a sequential manner. The design unpacks the flattened input vectors into two-dimensional arrays, performs matrix multiplication combinationally, and repacks the result. While the design is simple and parameterizable, it operates in a purely combinational fashion and does not yet incorporate pipelining or parallel MAC units. Future versions may include pipelined datapaths to achieve higher throughput and timing closure.

Listing 2.2: Full RTL listing for the parameterized matrix-multiplication accelerator

```
module matrix_multiplication #(parameter N = 4, width = 16) (
  input [0:N*N*width-1] a,
  input [0:N*N*width-1] b,
  output reg [0:2*N*N*width-1] y
);

  reg [width-1:0] a1[0:N-1][0:N-1];
  reg [width-1:0] b1[0:N-1][0:N-1];
  reg [2*width-1:0] y1[0:N-1][0:N-1];

  integer i, j, k;

  always @(a or b) begin
    // Unpack a and b
    for (i = 0; i < N; i = i + 1) begin
      for (j = 0; j < N; j = j + 1) begin
        a1[i][j] = a[(i*N + j)*width +: width];//part-select syntax
        b1[i][j] = b[(i*N + j)*width +: width];
      end
    end

    // Matrix multiplication
    for (i = 0; i < N; i = i + 1) begin
      for (j = 0; j < N; j = j + 1) begin
        y1[i][j] = 0;
        for (k = 0; k < N; k=k+1) begin
          y1[i][j] = y1[i][j] + (a1[i][k] * b1[k][j]);
        end
```

```
29        end
30      end
31
32      // Pack y
33      for (i = 0; i < N; i = i + 1) begin
34        for (j = 0; j < N; j = j + 1) begin
35          y[(i*N+j)*2*width +: 2*width] = y1[i][j];
36        end
37      end
38    end
39
40  endmodule
```

Implementation Notes

- **Parameterisation** The design uses two configurable parameters:
  - `N`: Sets the matrix dimension for square `N x N` multiplication.
  - `width`: Specifies the bit-width of each input element (e.g., 8 or 16 bits).

  This allows the design to be reused across a range of precision and throughput targets, without modifying internal logic.

- **Matrix Representation in Verilog** The two input matrices `a` and `b` are provided as flattened packed vectors. They are unpacked into 2D registers `a1[N][N]` and `b1[N][N]` using Verilog's part-select syntax:

  ```
  1  a1[i][j] = a[(i*N + j)*width +: width];
  ```

  This enables readable loop-based access and simplifies the matrix arithmetic logic.

- **Nested Loop Multiply-Accumulate (MAC)** The computation follows the canonical triple-loop matrix multiply algorithm:
  - Iterate over each row `i` of `a1` and each column `j` of `b1`.
  - For each output entry, iterate over `k` to accumulate `a1[i][k] * b1[k][j]`.

  In Verilog, these loops appear inside an `always_comb` block, which implies a combinational implementation. However, actual hardware synthesis behavior may vary depending on the target toolchain's treatment of large combinational logic.

- **Output Packing** Once computed, the result matrix `y1[N][N]` is flattened back into a single vector `y` using another nested loop and part-selects. The width of each output element is twice the input width to safely capture the product and sum: `reg [2*width -1:0] y1[N][N]`

- **Synthesis Considerations**
  - Although structurally valid, the fully combinational form may not scale well for large `N` due to adder tree depth.
  - Synthesis tools may issue warnings or infer unintended latches if loops are not fully bounded or written clearly.
  - For practical deployment on FPGA or ASIC, designers may need to rework the design to introduce explicit timing control (e.g., pipelining or handshaking).
- **Toward a Pipelined Version** Several improvements could be made for scalability and performance:
  - Insert registers between multiplication and accumulation to create a pipelined MAC datapath.
  - Replace the triple-loop with a controller FSM and a single MAC unit to save area at the cost of higher latency.
  - Use valid-ready or memory-mapped I/O interfaces to support streaming operands from an external controller or DMA engine.

  These changes require careful verification, as corner cases (e.g., matrix boundary or precision overflow) may introduce functional or timing issues.
- **Verification and Debugging**
  - The design is structured for easy unit testing with deterministic inputs and observable outputs.
  - The unpack/pack logic may require careful bit indexing during testbench setup, especially for large `N`.
  - Reference results from high-level software (e.g., NumPy, MATLAB) are suitable for comparison under simulation.

2.4 AlexNet Convolution Accelerator Design

Convolutional Neural Networks (CNNs) have become foundational in modern computer vision, enabling breakthroughs in image classification, object detection, and segmentation. One of the most influential architectures in this space is **AlexNet**, proposed by Krizhevsky et al. [14], which won the 2012 ImageNet competition by a significant margin. AlexNet introduced deep convolutional pipelines with ReLU activation, local response normalization, and max pooling, demonstrating the power of deep learning when combined with GPU acceleration.

To explore hardware-level acceleration of CNN inference, we implemented a standalone RTL design for a single convolutional layer inspired by AlexNet's first stage. This module

accepts streamed pixel data and filter coefficients, performs a full 11×11 convolution, applies ReLU activation, and finally reduces the result with $2{\times}2$ max pooling. While simplified compared to full AlexNet (which includes multiple layers, normalization, and overlapping pooling), our design reflects the core pipeline operations used in early-stage CNN inference.

Listing 2.3 shows the complete RTL for the `alex35_final` convolution accelerator. The design manages input pixel and filter memories, convolution operations, activation functions, and pooling computations using a finite-state machine (FSM). Pixel data ($35{\times}35$, 8-bit unsigned) and filter data ($11{\times}11$, 8-bit signed) are streamed into the respective internal memories. The accelerator then executes convolution to generate a $7{\times}7$ feature map (32-bit signed). Subsequently, ReLU activation is applied, and a final $3{\times}3$ feature map is produced through max pooling. FSM states clearly delineate these computational steps, ensuring predictable data flow and synchronization.

Listing 2.3: Full RTL listing for the AlexNet convolution accelerator

```
1  module alex35_final(
2      input clk,
3      input rst,
4      input [7:0] pixel_in,
5      input [7:0] filter_in,
6      input pixel_valid,
7      input filter_valid,
8      input go,
9      output reg [31:0] outv [0:8], // 3x3 max pool outputs
10     output reg data_ready
11 );
12     // Memory declarations
13     reg [7:0] aa [0:34][0:34];        // 35x35 pixel memory
14     reg signed [7:0] f [0:10][0:10]; // 11x11 filter
15     reg signed [31:0] conv_out [0:6][0:6]; // 32-bit convolution outputs
16     reg [31:0] relu_out [0:6][0:6];  // ReLU outputs
17
18     // Control signals
19     reg [15:0] cycle_count;
20     reg [4:0] x_conv, y_conv;
21     reg [3:0] state;
22
23     parameter IDLE = 0, LOAD_PIX = 1, LOAD_FILTER = 2,
24               CONV = 3, RELU = 4, POOL = 5;
25
26     // Control FSM
27     always @(posedge clk) begin
28         if (rst) begin
29             state <= IDLE;
30             data_ready <= 0;
31             cycle_count <= 0;
32             x_conv <= 0;
33             y_conv <= 0;
34         end else case(state)
```

```
35              IDLE: begin
36                  data_ready <= 0;
37                  if (pixel_valid) begin
38                      state <= LOAD_PIX;
39                      cycle_count <= 0;
40                  end else if (filter_valid) begin
41                      state <= LOAD_FILTER;
42                      cycle_count <= 0;
43                  end else if (go) begin
44                      state <= CONV;
45                      cycle_count <= 0;
46                      x_conv <= 0;
47                      y_conv <= 0;
48                  end
49              end
50
51              LOAD_PIX: begin
52                  aa[cycle_count/35][cycle_count%35] <= pixel_in;
53                  cycle_count <= cycle_count + 1;
54                  if (cycle_count == 1224) begin // 35x35=1225
55                      state <= IDLE;
56                      cycle_count <= 0;
57                  end
58              end
59
60              LOAD_FILTER: begin
61                  f[cycle_count/11][cycle_count%11] <= filter_in;
62                  cycle_count <= cycle_count + 1;
63                  if (cycle_count == 120) begin // 11x11=121
64                      state <= IDLE;
65                      cycle_count <= 0;
66                  end
67              end
68
69              CONV: begin
70                  if (cycle_count == 120) begin // 11x11 conv
71                      cycle_count <= 0;
72                      if (y_conv == 6) begin
73                          y_conv <= 0;
74                          x_conv <= x_conv + 1;
75                      end else begin
76                          y_conv <= y_conv + 1;
77                      end
78                      if (x_conv == 6 && y_conv == 6) state <= RELU;
79                  end else cycle_count <= cycle_count + 1;
80              end
81
82              RELU: begin
83                  relu_out[x_conv][y_conv] <= conv_out[x_conv][y_conv][31] ?
84                                              0 : conv_out[x_conv][y_conv];
```

```
85                  if (y_conv == 6) begin
86                      y_conv <= 0;
87                      x_conv <= x_conv + 1;
88                  end else y_conv <= y_conv + 1;
89                  if (x_conv == 6 && y_conv == 6) state <= POOL;
90              end
91
92              POOL: begin
93                  cycle_count <= cycle_count + 1;
94                  if (cycle_count == 8) begin
95                      data_ready <= 1;
96                      state <= IDLE;
97                  end
98              end
99          endcase
100     end
101
102     // Convolution
103     genvar i, j;
104     generate
105         for (i=0; i<7; i=i+1) begin: CONV_ROW
106             for (j=0; j<7; j=j+1) begin: CONV_COL
107                 reg signed [31:0] products [0:120];
108                 reg signed [31:0] accum;
109                 integer row, col;
110
111                 always @(posedge clk) begin
112                     if (state == CONV) begin
113                         row = cycle_count / 11;
114                         col = cycle_count % 11;
115                         products[cycle_count] <=
116                             $signed({24'd0, aa[i*4 + row][j*4 + col]}) *
117                             $signed({{24{f[row][col][7]}}, f[row][col]});
118                     end
119                 end
120
121                 always @(posedge clk) begin
122                     if (state == CONV) begin
123                         if (cycle_count == 0) accum <= products[0];
124                         else accum <= accum + products[cycle_count];
125                         if (cycle_count == 120)
126                             conv_out[i][j] <= accum+ products[120];
127                     end else begin accum <= 0; end
128                 end
129             end
130         end
131     endgenerate
132
133     // Max Pooling Unit
134     always @(posedge clk) begin
```

```
        if (state == POOL) begin : max_pooling
            integer px, py, dx, dy;
            integer x_base, y_base;
            reg [31:0] max_val;

            for (py = 0; py < 3; py = py + 1) begin
                for (px = 0; px < 3; px = px + 1) begin
                    x_base = px * 2;
                    y_base = py * 2;
                    max_val = 0;

                    for (dy = 0; dy < 3; dy = dy + 1) begin
                        for (dx = 0; dx < 3; dx = dx + 1) begin
                            if ((x_base + dx < 7) && (y_base + dy < 7))
                                begin
                                if (relu_out[y_base + dy][x_base + dx] >
                                    max_val) begin
                                    max_val = relu_out[y_base + dy][x_base
                                        + dx];
                                end
                            end
                        end
                    end
                    outv[py*3 + px] <= max_val;
                end
            end
        end
    end
endmodule
```

Implementation Notes

- **On-chip Memories**
  - *Pixel Store (aa):* Implemented as a two-dimensional `reg [7:0] aa [0:34][0:34]`.
    - *Synthesis mapping.* On FPGAs, synthesis may infer block/distributed RAM for such arrays; on ASIC flows, an array-of-regs will become flip-flops *unless* memory inference/mapping to a foundry SRAM macro is enabled (or an SRAM macro is instantiated).
    - Row-major addressing: addr $=$ `row` $\times 35 +$ `col` (e.g., (`row`$\ll 5) + ($`row`$\ll 1) +$ `row` $+$ `col`), so a simple adder-multiplier (or shift-add) network drives the address port.
  - *Filter Store (f):* 11×11 signed coefficients held in an identical fashion. Because only one filter is used, no banking is required.
  - *Intermediate Buffers:* `conv_out` and `relu_out` are true 7×7 register files; for 49 32-bit words, dedicated flip-flops are cheaper than instantiating another SRAM.

- **Synthesizable `for` loops: quick notes**
  - *Generate-time replication.* In `generate` blocks (with a `genvar`), the loop elaborates at compile time and creates N copies of hardware. For the 7x7 convolution nest:

```
generate
  for (i = 0; i < 7; i = i + 1)
    for (j = 0; j < 7; j = j + 1)
      ...  // 49 instances
endgenerate
```

    the design contains 49 independent MAC "cores", one per output pixel, operating in parallel.
  - *Combinational loops.* In `always_comb` (or `always @(*)`), a `for` loop describes combinational logic and is unrolled by synthesis; it is not a run-time iterator. Loop bounds must be static.
  - *Clocked loops.* In `always_ff`, avoid multiple `<=` to the same register inside the loop. Compute into a temporary with blocking `=`, then perform a single non-blocking update:

```
always_ff @(posedge clk) begin
  automatic logic [W:0] acc = '0;
  for (int k = 0; k < NUM; k = k + 1) acc = acc + prod[k];
  sum_q <= acc;  // one <= per register per cycle
end
```

  - Provide defaults in `always_comb` to avoid latch inference, and avoid data-dependent loops (`while`, `break`, `continue`).
- **Cycle Budget** (current non-pipelined schedule)

Stage	Formula	Cycles
Pixel load	35×35	1225
Filter load	11×11	121
Convolution	11×11 (all 49 cores in parallel)	121
ReLU	7×7 sequential write-back	49
Max pool	9 windows, 1 cycle/window	9
Total		**1525**

The FSM's `cycle_count` register implements the above latency bookkeeping; each state terminates when its precomputed limit is reached.

- **Why 121 Convolution Cycles?** Each MAC instance needs every element of the 11×11 receptive field. A single global counter (`cycle_count`) broadcasts `row/col` indices to all 49 cores, so the entire 7×7 tile finishes after $11 \times 11 = 121$ clocks.

- **Future Pipelining Road-map**

  1. *Stage-level overlap (double buffering).* While one image tile is in ReLU, the next tile can already start convolution provided two independent `conv_out/relu_out` buffers are available.
  2. *Spatial pipeline.* Insert a register slice after the multiplier array so that product generation and accumulation occupy separate pipeline stages; the critical path shrinks from "multiplier + adder" to one of the two.
  3. *Activation/Pool streaming.* Re-implement ReLU and 2×2 pooling as combinational taps on the accumulator FIFO; this hides both phases under the convolution latency but requires careful hazard checks (partial sums must be final before pooling).
  4. *Hazard pitfalls.*
     - Write-after-read on `relu_out` if pooling starts too early.
     - Increased BRAM port pressure: overlapping stages may need two read ports (one for new products, one for pooling).

  A two-deep pipeline (CONV ∥ ReLU) can give about $1.4\times$ (up to $1.48\times$ if pooling is also overlapped) throughput but costs an extra 49 register words and a second set of pixel addresses; deeper pipelines show diminishing returns because pooling consumes only nine results.

- **ASIC Timing Snapshot** With the current flat pipeline the longest path is "multiplier $\rightarrow$ adder $\rightarrow$ accumulator register", closing at 6 ns (166 MHz). Splitting after the multiplier is expected to reduce the path to about 3 ns, enabling 330 MHz at the cost of one extra register layer.

2.5 AHB-SRAM Controller Design

Modern system-on-chip (SoC) designs rely heavily on standardized bus protocols to interconnect processing cores, memory blocks, and peripheral components. The Advanced Microcontroller Bus Architecture (AMBA), developed by Arm, is a widely adopted open specification that provides such an interconnect framework. It includes a family of bus protocols such as APB (Advanced Peripheral Bus), AHB (Advanced High-performance Bus), and AXI (Advanced eXtensible Interface), each tailored for different bandwidth and complexity requirements.

The AMBA AHB-Lite protocol is a streamlined subset of AHB designed for single-master environments. It provides a pipelined bus structure that supports burst and non-burst transfers, separate address and data phases, and built-in handshaking to ensure timing-safe communication. AHB-Lite is especially suitable for microcontroller-style designs where a single bus master (e.g., a CPU or DMA) communicates with multiple memory-mapped slave devices such as SRAM, flash, or I/O controllers.

In this context, we developed an AHB-compliant SRAM controller that serves as a memory-mapped slave device on the AHB bus. The controller supports four operational modes: **IDLE**, **READ**, **WRITE**, and **BIST** (Built-In Self-Test). It interfaces with both the AHB master (via a standard slave interface) and a backend SRAM block that is split across two memory banks. The controller handles all AHB protocol signals, including address decoding, data multiplexing, and handshaking, while generating SRAM chip select, address, and data signals based on the transaction type.

Listing 2.4 shows the complete RTL for the `ahb_slave_if_final` module, which serves as the AHB-compliant slave interface in a multi-mode SRAM controller. The full design includes address decoding, handshaking logic, and mode selection logic to support four operational modes: IDLE, READ, WRITE, and BIST. This controller was written in Verilog, synthesized with Synopsys Design Compiler, and simulated with VCS and Verdi.

Listing 2.4: AHB-compliant slave interface module for SRAM controller

```
1  module ahb_slave_if (
2      // Clock & Reset
3      input  wire          hclk,
4      input  wire          hresetn,
5
6      // AHB-Lite Interface Inputs
7      input  wire          hsel,
8      input  wire          hwrite,
9      input  wire          hready,
10     input  wire [2:0]    hsize,
11     input  wire [2:0]    hburst,
12     input  wire [1:0]    htrans,
13     input  wire [31:0]   hwdata,
14     input  wire [31:0]   haddr,
15
16     // SRAM Read Data Inputs
17     input  wire [7:0]    sram_q0,
18     input  wire [7:0]    sram_q1,
19     input  wire [7:0]    sram_q2,
20     input  wire [7:0]    sram_q3,
21     input  wire [7:0]    sram_q4,
22     input  wire [7:0]    sram_q5,
23     input  wire [7:0]    sram_q6,
24     input  wire [7:0]    sram_q7,
25
26     // AHB-Lite Outputs
27     output wire          hready_resp,
```

```
28      output wire [1:0]   hresp,
29      output wire [31:0]  hrdata,
30
31      // SRAM Control Outputs
32      output wire         sram_w_en,
33      output wire [12:0]  sram_addr_out,
34      output wire [31:0]  sram_wdata,
35      output wire [3:0]   bank0_csn,
36      output wire [3:0]   bank1_csn
37  );
38
39    // -------------------------------------------------------------
40    // Parameters & Internal Declarations
41    // -------------------------------------------------------------
42    parameter IDLE   = 2'b00,
43              BUSY   = 2'b01,
44              NONSEQ = 2'b10,
45              SEQ    = 2'b11;
46
47    wire        gated_clk;
48    wire        idle_signal = !(hsel && (htrans != IDLE));
49    assign      gated_clk = idle_signal ? 1'b0 : hclk;
50
51    // Internal registers for pipelining
52    reg         hwrite_r;
53    reg [2:0]   hsize_r;
54    reg [2:0]   hburst_r;
55    reg [1:0]   htrans_r;
56    reg [31:0]  haddr_r;
57
58    reg [31:0]  hwdata_p;
59    reg [12:0]  sram_addr_p;
60    reg [3:0]   sram_csn;
61
62    // Internal control logic
63    wire [1:0]  haddr_sel = sram_addr[1:0];
64    wire [1:0]  hsize_sel = hsize_r[1:0];
65    wire        sram_csn_en, sram_write, sram_read, bank_sel;
66    wire [15:0] sram_addr = haddr_r[15:0];
67    wire [31:0] sram_data_out;
68
69    // -------------------------------------------------------------
70    // Pipelined Data Capture
71    // -------------------------------------------------------------
72    always @(posedge gated_clk or negedge hresetn) begin
73      if (!hresetn) begin
74        sram_addr_p <= 13'b0;
75        hwdata_p    <= 32'b0;
76      end else if (sram_write) begin
77        sram_addr_p <= sram_addr_out;
```

```
78        hwdata_p     <= hwdata;
79      end
80    end
81
82    assign sram_wdata = hwdata_p;
83
84    // ------------------------------------------------------------
85    // Output Assignments
86    // ------------------------------------------------------------
87    assign hready_resp = 1'b1;
88    assign hresp       = (htrans == IDLE || hburst == 3'b000) ? 2'b00 : 2'
        b01;
89    assign hrdata      = sram_data_out;
90
91    assign sram_data_out = bank_sel ?
92                           {sram_q3, sram_q2, sram_q1, sram_q0} :
93                           {sram_q7, sram_q6, sram_q5, sram_q4};
94
95    assign sram_write    = ((htrans_r == NONSEQ) || (htrans_r == SEQ)) &&
        hwrite_r;
96    assign sram_read     = ((htrans_r == NONSEQ) || (htrans_r == SEQ)) && !
        hwrite_r;
97    assign sram_w_en     = !sram_write;
98
99    assign sram_addr_out = sram_addr[14:2];
100
101   assign sram_csn_en   = sram_write || sram_read;
102   assign bank_sel      = sram_csn_en && (sram_addr[15] == 1'b0);
103
104   assign bank0_csn     = (bank_sel) ? sram_csn : 4'b1111;
105   assign bank1_csn     = (!bank_sel) ? sram_csn : 4'b1111;
106
107   // ------------------------------------------------------------
108   // SRAM Chip Select Decoding
109   // ------------------------------------------------------------
110   always @(*) begin
111     case (hsize_sel)
112       2'b10: sram_csn = 4'b0000; // Word
113       2'b01: sram_csn = haddr_sel[1] ? 4'b0011 : 4'b1100; // Halfword
114       2'b00: begin // Byte
115         case (haddr_sel)
116           2'b00: sram_csn = 4'b1110;
117           2'b01: sram_csn = 4'b1101;
118           2'b10: sram_csn = 4'b1011;
119           2'b11: sram_csn = 4'b0111;
120           default: sram_csn = 4'b1111;
121         endcase
122       end
123       default: sram_csn = 4'b1111;
124     endcase
```

```
  end

  // ---------------------------------------------------------
  // AHB Control Signal Pipelining
  // ---------------------------------------------------------
  always @(posedge gated_clk or negedge hresetn) begin
    if (!hresetn) begin
      hwrite_r  <= 1'b0;
      hsize_r   <= 3'b000;
      hburst_r  <= 3'b000;
      htrans_r  <= 2'b00;
      haddr_r   <= 32'b0;
    end else if (hsel && hready) begin
      hwrite_r  <= hwrite;
      hsize_r   <= hsize;
      hburst_r  <= hburst;
      htrans_r  <= htrans;
      haddr_r   <= haddr;
    end
  end

endmodule
```

Implementation Notes

- **Modular Structure** The AHB-SRAM controller consists of:
  - `ahb_slave_if_final`: handles AHB handshaking and transfer decoding.
  - `sram_core`: manages address generation, enable logic, and data multiplexing.
  - `sram_bist`: executes memory self-test patterns and result checks.
  - `bt_sram_8kx8`: SRAM macro model used for simulation.
- **AHB Protocol Handling** The slave interface adheres to the AMBA AHB-Lite protocol and supports the following signals:
  - Address phase: `hsel, haddr, htrans, hwrite, hsize`
  - Data phase: `hwdata, hrdata, hready, hready_resp`

  The module decodes burst and single transfers and generates valid read/write enables accordingly.
- **FSM-Based Control** A simple FSM governs the mode transitions between IDLE, READ, WRITE, and BIST. Each mode asserts corresponding enables:
  - `write_en_o`, `read_en_o` control SRAM access.

- `bist_en` enables internal pattern testing.
- `addr_o` and `wdata_o` are valid only in the WRITE mode.

- **Supported AHB Transfer Types**

Transfer Type	Behavior
IDLE	No transfer; controller stays in low-power mode.
NONSEQ	Initiates a fresh read or write transaction.
SEQ	Used in burst accesses to auto-increment address.
BUSY	No state change; address and data are held stable.

2.6 Summary

This chapter presented four RTL designs that collectively trace the evolution from basic control logic to full-scale compute and memory subsystems. These designs served not only as technical implementations but also as learning platforms to master modularity, scalability, and verification-readiness in hardware development.

The FSM design emphasized clear clock-domain separation and formal architectural style. It enforced clean state transitions and enabled effective simulation and coverage analysis.

The matrix multiplication engine introduced scalable datapath techniques, including generate loops, part-select logic, and parameterization. It established a reusable compute kernel and introduced the tradeoffs between performance, area, and hardware regularity.

The AlexNet-style CNN accelerator brought together memory arrays, multiply-accumulate pipelines, activation units, and control FSMs into a single structured datapath. Despite its non-pipelined nature, the design exposed all intermediate signals and stages, preparing the ground for future enhancements like streaming computation and deeper parallelism.

Finally, the AHB-SRAM controller illustrated integration with a standard SoC bus protocol and detailed memory timing coordination. It implemented address pipelining, handshake logic, and memory bank selection, verifying functional correctness through waveform-based debugging and backend timing closure.

These designs now underpin the verification methodology discussed in Chapter 3, serve as input netlists for synthesis and physical design in Chapter 4, and provide test vehicles for security analysis and AI-assisted automation in Chapters 6 and 7.

Chapter 3

Hardware Verification with Synopsys VCS

This chapter presents a practical and reproducible methodology for verifying RTL designs using the *Synopsys VCS and Verdi* simulation and debugging toolchain. It covers compilation, simulation, waveform inspection, coverage analysis, and runtime profiling—all essential steps in ensuring the functional correctness and completeness of a digital hardware design.

All workflows described here have been validated using the College of Engineering lab container on the UC Davis infrastructure, but they are equally portable to any local or cloud-based installation of the Synopsys tools once environment variables are configured. Each example in this chapter uses real RTL modules developed in Chapter 2, including the FSM and matrix multiplier.

To further streamline this process, I also introduce custom automation scripts, including a portable Makefile, a batch test runner, and a coverage extraction utility—that encapsulate the simulation flow and facilitate batch testing and metric collection. These tools are designed to scale across modules and testbenches and were used consistently throughout the thesis.

3.1 VCS at a Glance

Synopsys VCS ("Verilog Compiler Simulator") is a *high-performance, mixed-language simulation and verification environment*. It allows designers to

- **simulate** RTL code before synthesis,
- **verify** functional correctness with self-checking test-benches,
- **debug** signal activity interactively in Verdi, and
- **analyze** test completeness through built-in coverage and profiling engines.

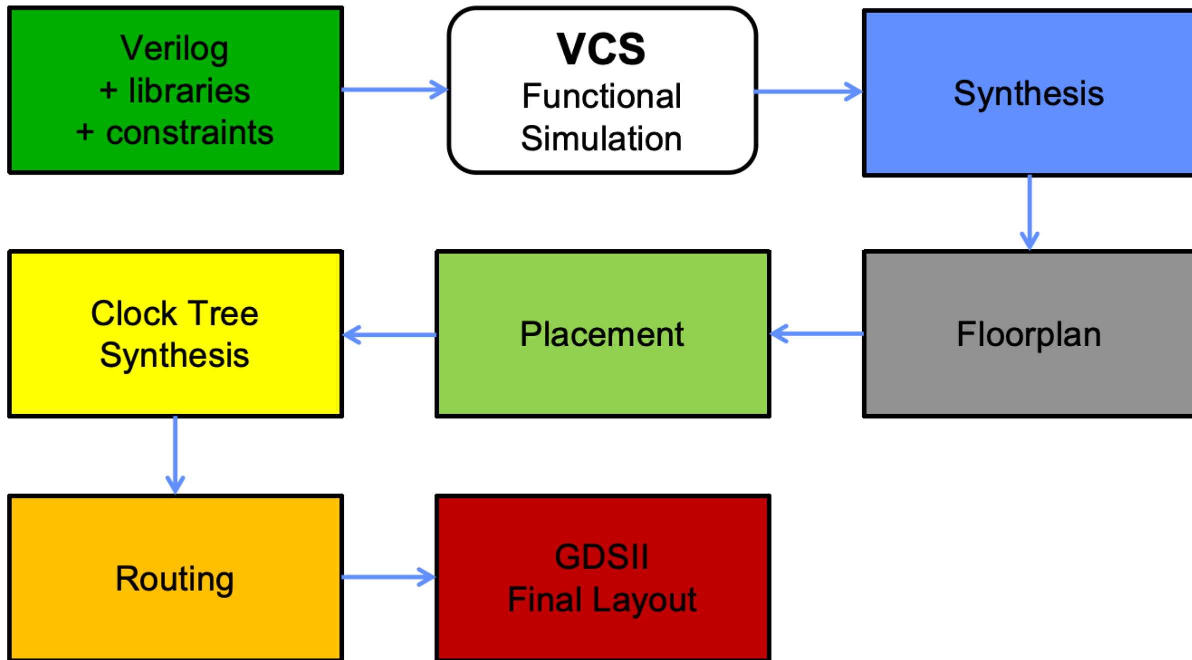

Figure 3.1: Why VCS is central to the digital-IC design flow

VCS supports two usage models [13, 1.2]: a *2-step "compile–simulate" flow* for pure-Verilog projects and a *3-step "analyze–compile–simulate" flow* for mixed-HDL designs. In this thesis, we restrict ourselves to the lean 2-step flow because all examples are SystemVerilog RTL.

3.2 Environment Setup

Connecting to the Lab Server

1. Connect to the College-of-Engineering VPN.[1]

2. Launch an X11 session on the lab server (GUI forwarding is required for Verdi):

```
1 ssh -X <username>@rubichest.ece.ucdavis.edu
```

3. Enter the Apptainer container that holds the toolchain:

```
1 apptainer shell --bind /opt /opt/synopsys/rocky.sif
```

[1] https://vpn.engineering.ucdavis.edu

Tool Variables (checked once)

The server's shell initialization file already exports the correct paths, but they are listed here for completeness [13, 1.4]:

```
export VCS_HOME=/opt/synopsys/vcs/W-2024.09-1
export VERDI_HOME=/opt/synopsys/verdi/W-2024.09-1
export PATH=$VCS_HOME/bin:$VERDI_HOME/bin:$PATH
```

3.3 Compilation and Simulation

Step 1 - Compile

```
vcs -full64 -sverilog -debug_access+all \
    tb_fsm.sv fsm.sv -l compile.log
```

Key switches:

- `-full64`: force 64-bit executables.
- `-sverilog`: enable SystemVerilog constructs.
- `-debug_access+all`: emit a KDB database for Verdi.:contentReference[oaicite:3]index=3

The compiler produces `simv` (the simulation binary) and `simv.daidir/` (kernel database).

Step 2 - Simulate

```
./simv                # plain batch run
./simv -gui           # interactive Verdi session
./simv -cm line+tgl+cond+fsm+branch -cm_name smoke
```

The final variant turns on line, toggle, condition, FSM, and branch coverage metrics so they are captured into the default `simv.vdb` database.

3.4 Interactive Debug with Verdi

Verdi can be launched either *post-simulation* (load FSDB/KDB files) or *interactive* (run the kernel under the GUI).

Typical sequence [13, Chap. 2]:

1. Compile with `-debug_access+all -kdb`
2. Simulate and dump `fsm.fsdb`.

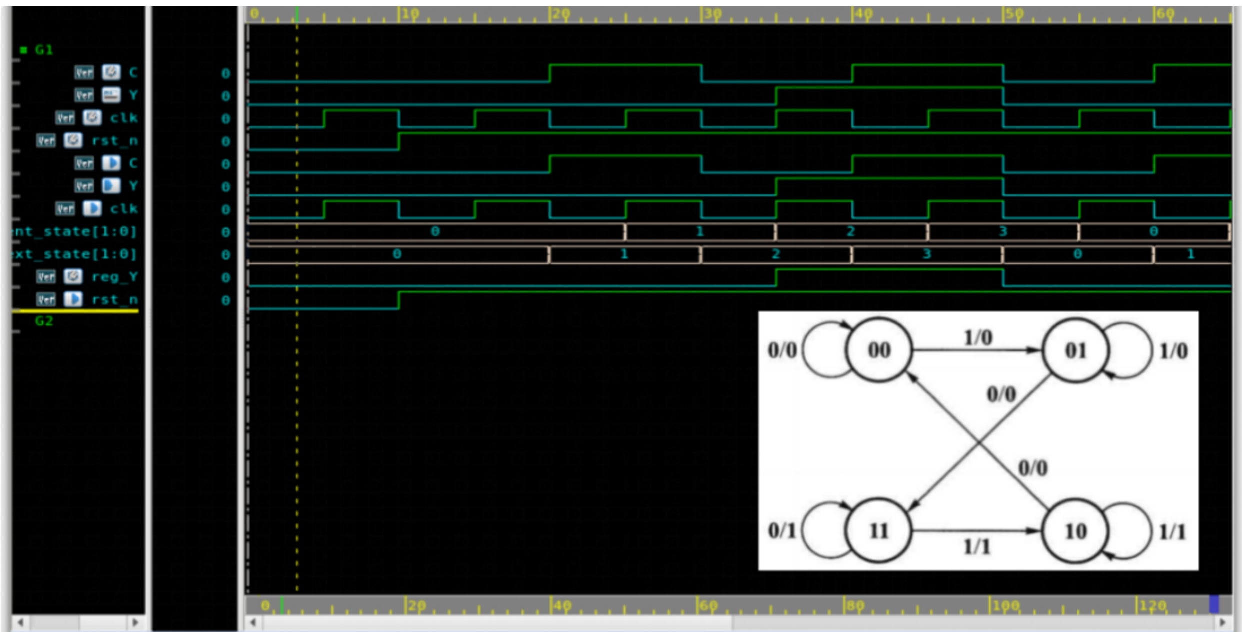

Figure 3.2: Verdi nWave session showing the four-state FSM waveform [15]

3. Open Verdi:

```
verdi -ssf fsm.fsdb -dbdir simv.daidir
```

4. Use the *Hierarchy*, *Signal*, and *nWave* panes to cross-probe code and waveform (Fig. 3.2).

3.5 Coverage Analysis

After a coverage-enabled run, generate an HTML dashboard [13, 3.8]:

```
urg -dir simv.vdb -report coverage_report
```

Open `coverage_report/dashboard.html` in any browser to inspect metrics such as *LINE*, *TOGGLE*, *FSM*, and *BRANCH*. Figure 3.3 shows a sample dashboard for the improved `tb_fsm_new.sv` test-bench.

3.6 Runtime Profiling

Beyond functional correctness, it is often important to understand which parts of a testbench or design consume the most simulation time. Synopsys VCS supports this capability through its `-simprofile` switch, which instruments the simulation kernel and generates detailed timing breakdowns.

SYNOPSYS®

Dashboard

dashboard | hierarchy | modlist | groups | tests | asserts

Date: Wed Mar 26 02:59:19 2025
User: yuyang07
Version: W-2024.09-1
Command line: urg -dir simv.vdb -report coverage_report
Number of tests: 1

Total Coverage Summary

SCORE	LINE	COND	TOGGLE	FSM	BRANCH
91.79	100.00	100.00	92.31	66.67	100.00

Hierarchical coverage data for top-level instances

SCORE	LINE	COND	TOGGLE	FSM	BRANCH	NAME
91.79	100.00	100.00	92.31	66.67	100.00	tb_fsm

Total Module Definition Coverage Summary

SCORE	LINE	COND	TOGGLE	FSM	BRANCH
91.79	100.00	100.00	92.31	66.67	100.00

Figure 3.3: Code-coverage dashboard generated by URG [15]

```
vcs -full64 -simprofile -debug_access+all \
    tb_mm.sv matrix_multiplication.sv -o simv
./simv +simprofile=time
profrpt -view time_all simprofile_dir -output profileReport
```

The first command compiles the design and enables runtime profiling instrumentation. The second runs the simulation with profiling enabled, and the third uses `profrpt` to generate an HTML report. The resulting output categorizes time spent in RTL execution, SystemVerilog testbench code, constraint-solving engines, and more [13, Chap. 4]. This is especially helpful when optimizing testbenches or when identifying whether a design block or a verification environment is the simulation bottleneck.

3.7 Workflow Checklist

Table 3.1 summarizes the streamlined two-step verification flow adopted throughout this thesis. This methodology—documented and refined by the author—encapsulates the core workflow for simulation, coverage collection, and performance profiling using Synopsys VCS and Verdi. It enables rapid iteration on both RTL correctness and testbench effectiveness.

This flow serves as the backbone for verifying all RTL modules discussed in Chapter 2.

```
==========================================================================================
                                   Time Summary View
==========================================================================================
  Component                                                          Percentage
------------------------------------------------------------------------------------------
 UCLI                                                                   69.23%
 KERNEL                                                                 15.38%
   License                                                               7.69%
   Hsim_Elab                                                             3.85%
 VERILOG                                                                 7.69%
   Module                                                                7.69%
 PLI/DPI/DirectC                                                         7.69%
------------------------------------------------------------------------------------------
  TOTAL                                                                100.00%
__________________________________________________________________________________________

==========================================================================================
                                   Time Instance View
==========================================================================================
       Instance            %TotalTime      Module/Program  Source
                            incl|excl       /Architecture
------------------------------------------------------------------------------------------
tb_matrix_multiplication     7.69|7.69     tb_matrix_multi  /home/yuyang07/snops_workdir/vcs/tb_matrix_multiplic
                                                 plication  ation.sv:1
  NoName {line:18}(Initia    7.69|7.69                  --  /home/yuyang07/snops_workdir/vcs/tb_matrix_multiplic
  l)                                                        ation.sv:18
__________________________________________________________________________________________
```

Figure 3.4: Simulation time breakdown using `simprofile` and `profrpt` [15]

Table 3.1: Minimal two-step VCS flow for RTL verification.

Stage	**Key Command(s) and Options**
Compile	`vcs -full64 -sverilog -debug_access+all <tb> <rtl>`
Simulate	`./simv [-gui] [-cm line+tgl+cond+fsm+branch] [+simprofile=time]`
Post-process	Coverage: `urg -dir simv.vdb -report report/` Profiling: `profrpt -view time_all simprofile_dir`

It is fully compatible with waveform-based debugging (Figure 3.2), coverage dashboards (Figure 3.3), and timing/profile analysis (Figure 3.4). By consolidating simulation, coverage, and profiling into a unified flow, the workflow ensures consistency and reproducibility across multiple designs and testbenches.

3.8 Build Automation Scripts

To automate and standardize the simulation, coverage, and reporting flow across multiple testbenches, I developed a minimal set of build scripts. These include a portable Makefile for compilation and simulation, a batch runner for test iteration, and a post-processing script for coverage aggregation. All scripts were used consistently across aforementioned RTL projects.

Makefile for Simulation and Coverage

The top-level `Makefile` supports Verilog/SystemVerilog compilation with Synopsys VCS, enabling both interactive and coverage-enabled runs. It accepts a test name suffix via the `TEST_NAME` variable and produces isolated coverage reports for each test instance. It also enables conditional waveform dumping, integrates with URG for coverage reporting, and supports customizable module lists and runtime flags. Outputs are placed in dedicated test directories to avoid conflicts and preserve results.

Listing 3.1: Makefile for VCS simulation and coverage automation

```
# - Configuration -
TOP_MODULE = matrix_multiplication
SRC = tb_$(TOP_MODULE).sv $(TOP_MODULE).sv
OUT = simv
FSDB_FILE = $(TOP_MODULE).fsdb

# - Flags -
VCS_FLAGS = -full64 -sverilog -debug_access+all
COV_FLAGS = -cm line+tgl+cond+fsm+branch
PROF_FLAGS = -simprofile

# - Default Target -
all: clean compile simulate

# - Compilation -
compile:
  vcs $(VCS_FLAGS) $(SRC) -o $(OUT)

# - Simulation -
simulate:
  ./$(OUT)

# - Interactive Simulation with Verdi GUI -
gui:
  ./$(OUT) -gui

# - Open FSDB via Verdi -
debug:
  verdi -ssf $(FSDB_FILE)

# - Coverage -
coverage:
  vcs $(VCS_FLAGS) $(COV_FLAGS) $(SRC) -o $(OUT)
  ./$(OUT) $(COV_FLAGS) -cm_name test_coverage
  urg -dir $(OUT).vdb -format both -report coverageReport

# - View Coverage report with Verdi GUI -
coverage_gui:
  verdi -cov -covdir simv.vdb
```

```
40
41 # - Time Profiling Only -
42 profile_time:
43   vcs $(VCS_FLAGS) $(PROF_FLAGS) $(SRC) -o $(OUT)
44   ./$(OUT) +simprofile=time
45   rm -rf profileReport
46   profrpt -view time_all -filter 0 simprofile_dir -output profileReport
47
48 # - Memory Profiling Only -
49 profile_mem:
50   vcs $(VCS_FLAGS) $(PROF_FLAGS) $(SRC) -o $(OUT)
51   ./$(OUT) +simprofile=mem
52   rm -rf profileReport
53   profrpt -view mem_all -filter 0 simprofile_dir -output profileReport
54
55 # - Clean-up -
56 clean:
57   rm -rf $(OUT) *.vpd *.fsdb *.log simv.daidir csrc \
58          verdiLog ucli.key $(OUT).vdb simprofile_dir* \
59          coverageReport* urgReport* profileReport* \
60          *.saif *.rc *.json *.html *.txt *.xml *.vdb \
61          vdCov* novas.*
```

run.sh: Batched Simulation Driver

This shell script loops over multiple test indices and invokes the Makefile's `coverage_new` target. Each run uses a unique `TEST_NAME` to produce a separate coverage database. This is particularly helpful when running randomized or fuzzed testbenches.

Listing 3.2: Automated test loop for batch simulation

```
1 #!/bin/bash
2
3 for i in $(seq 1 20); do
4   echo "     Running VCS with TEST_COUNT = $i"
5   make coverage_new TEST_NAME=$i
6 done
7
8 echo "    All 20 test case simulations completed."
```

extract_coverage.sh: Coverage Aggregation Script

Once all simulations complete, this script collects metrics (e.g., score, line, toggle, FSM coverage) from all generated coverage reports. It extracts and consolidates the results into a CSV file, enabling structured post-analysis and visualization.

Listing 3.3: Script to extract coverage data from all test reports

```
#!/bin/bash

OUTPUT_CSV="all_coverage_summary.csv"
echo "TEST_NAME,SCORE,LINE,COND,TOGGLE,FSM,BRANCH" > "$OUTPUT_CSV"

# Loop through all dashboard.txt files under coverageReport_* folders
find . -path "*/coverageReport_*/dashboard.txt" | while read dashboard; do
    # Extract test name (e.g., coverageReport_fsm     fsm)
    full_test_name=$(grep -oP 'report\s+\KcoverageReport_[^ ]+' "$dashboard")
    TEST_NAME=${full_test_name#coverageReport_}

    # Extract coverage numbers from the first matching line
    COVERAGE_LINE=$(grep -m 1 '^[[:space:]]*[0-9]' "$dashboard")
    SCORE=$(echo "$COVERAGE_LINE" | awk '{print $1}')
    LINE=$(echo "$COVERAGE_LINE" | awk '{print $2}')
    COND=$(echo "$COVERAGE_LINE" | awk '{print $3}')
    TOGGLE=$(echo "$COVERAGE_LINE" | awk '{print $4}')
    FSM=$(echo "$COVERAGE_LINE" | awk '{print $5}')
    BRANCH=$(echo "$COVERAGE_LINE" | awk '{print $6}')

    # Output one row to CSV
    echo "$TEST_NAME,$SCORE,$LINE,$COND,$TOGGLE,$FSM,$BRANCH" >> "$OUTPUT_CSV"
done

# Sort the file numerically by test number (test1, test2, ..., test20)
(head -n 1 "$TEMP_CSV" && tail -n +2 "$TEMP_CSV" | sort -t_ -k2n) > "$OUTPUT_CSV"

rm "$TEMP_CSV"

echo "    All coverage summaries saved to $OUTPUT_CSV"
```

These automation tools ensure reproducibility and consistent workflow across all simulation-driven experiments in this thesis. They are portable, extensible, and were used throughout Chapter 3 in validating all RTL modules.

Chapter 4

ASIC Physical Design and EDA Toolchains

Physical design bridges the gap between high-level hardware description and physical layout suitable for fabrication. Once a digital block is described and verified at the RTL level, Electronic Design Automation (EDA) tools convert it into a gate-level implementation, optimize for timing, area, and power, and finally produce a GDSII layout for tape-out. In this chapter, we explore this transformation using both open-source and commercial flows.

We begin with a hands-on walkthrough of the open-source **OpenLane** framework [1], which performs full RTL-to-GDSII layout using the Sky130 process design kit (PDK) [16]. Two representative RTL designs—a RISC-V core `picorv32` [17] and a custom scratchpad memory `spm` are synthesized and routed using OpenLane, with configuration details and layout results fully documented.

In addition, we introduce the industry-standard **Synopsys Design Compiler (DC)** to perform RTL-level logic synthesis on a third compute-heavy design, `alex35_final`. Design Compiler offers fine-grained analysis of critical paths, area utilization, and sequential complexity, and is commonly used for signoff synthesis in commercial toolchains. By comparing OpenLane and DC outputs, this chapter provides a comprehensive overview of modern ASIC design pipelines for both academic prototyping and industry-aligned flows.

4.1 RTL-to-GDSII: Conceptual Overview

The ASIC design flow transforms RTL code into a manufacturable layout through the following canonical stages:

- **Logic Synthesis**: Translates SystemVerilog/Verilog into a gate-level netlist using standard-cell libraries from a technology PDK.
- **Floorplanning**: Defines die size, aspect ratio, IO pin locations, and power distribution network (PDN) structure.

- **Placement**: Legally places each gate to minimize area and wire congestion.
- **Clock Tree Synthesis (CTS)**: Balances the clock signal distribution by inserting buffers and meeting skew targets.
- **Routing**: Physically connects all gates and IO using metal layers defined by the PDK.
- **Signoff Checks**: Performs design rule checking (DRC), layout versus schematic (LVS) checking, and parasitic extraction.
- **GDSII Export**: Writes a final `.gds` file that encodes mask layers for fabrication.

4.2 OpenLane Flow

OpenLane is an automated digital synthesis and layout toolchain built on top of tools such as Yosys, Magic, TritonRoute, and KLayout. It is designed to support the Sky130 open-source PDK provided by Google and SkyWater.

OpenLane's flow consists of approximately twenty stages, grouped into:

- **Synthesis**: Parses RTL, applies constraints, and generates a gate-level netlist.
- **Floorplan + PDN**: Allocates die area and generates a power grid.
- **Placement and Optimization**: Inserts logic, reorders for timing, performs congestion checks.
- **Routing and DRC**: Connects the design and validates it for fabrication.
- **Signoff**: Includes STA, parasitic extraction, GDS export, and report generation.

The OpenLane flow is controlled by a `config.tcl` file, which sets global and design-specific parameters.

4.3 Case Study 1: picorv32 RISC-V Core

The `picorv32` design is a compact, open-source RISC-V core developed by Clifford Wolf. Its well-documented RTL and parameterization make it suitable for exploring OpenLane's synthesis capabilities.

Configuration

The physical design flow in OpenLane is driven by a project-specific configuration file written in TCL. This script sets parameters for synthesis, floorplanning, placement, clocking, routing, and physical verification stages. These include constraints such as target clock period, I/O pin mapping, PDN grid pitch, cell density, and layer preferences.

Listing 4.1 shows the configuration file used to generate the GDSII layout for `picorv32` using the Sky130 PDK. Notable settings include a 10 ns clock period, a manually defined pin order, and fine-grained power distribution network (PDN) parameters. These constraints help ensure timing closure, compact area utilization, and manufacturability in SkyWater's open PDK environment.

Listing 4.1: OpenLane configuration file for `picorv32`.

```
1  # Design Name
2  set ::env(DESIGN_NAME) "picorv32"
3
4  # Path to the Verilog files
5  set ::env(VERILOG_FILES) "/Users/kevin/Desktop/OpenLane/designs/picorv32/
       src/picorv32.v"
6
7  # Synthesis settings
8  set ::env(SYNTH_STRATEGY) "DELAY 1"
9
10 # Define missing environment variable
11 set ::env(SYNTH_BUFFERING) 1
12
13 # Clock settings
14 set ::env(CLOCK_PORT) "clk"
15 set ::env(CLOCK_PERIOD) "10"
16
17 # I/O and Power Grid
18 set ::env(FP_IO_VPITCH) 160
19 set ::env(FP_IO_HPITCH) 160
20 set ::env(FP_PDN_VPITCH) 240
21 set ::env(FP_PDN_HPITCH) 240
22
23 # Placement settings
24 set ::env(PL_TARGET_DENSITY) 0.6
25 set ::env(GLB_RT_MAXLAYER) 5
26 set ::env(CELL_PAD) 2
27 set ::env(VT_FILL_CELL) "sky130_fd_sc_hd__fill_2"
28 set ::env(GRT_ALLOW_CONGESTION) 0
29
30 # Physical design
31 set ::env(PL_BASIC_PLACEMENT) 0
32 set ::env(PL_PADDING) 6
33
34
35 # PDK and standard cells
```

```
set ::env(PDK_ROOT) "/Users/kevin/.volare"
set ::env(PDK) "sky130A"

# Custom SDC files
set ::env(PNR_SDC_FILE) "/Users/kevin/Desktop/OpenLane/designs/picorv32/
    src/pnr.sdc"
set ::env(SIGNOFF_SDC_FILE) "/Users/kevin/Desktop/OpenLane/designs/
    picorv32/src/signoff.sdc"

# Library for synthesis
set ::env(LIB_SYNTH) "/Users/kevin/.volare/sky130A/libs.ref/
    sky130_fd_sc_hd/lib/sky130_fd_sc_hd__tt_025C_1v80.lib"
```

GDSII Output

The final layout is exported as `picorv32.gds`, visualized using KLayout or Magic. The core area shows well-placed logic, distributed standard cells, and aligned PDN tracks.

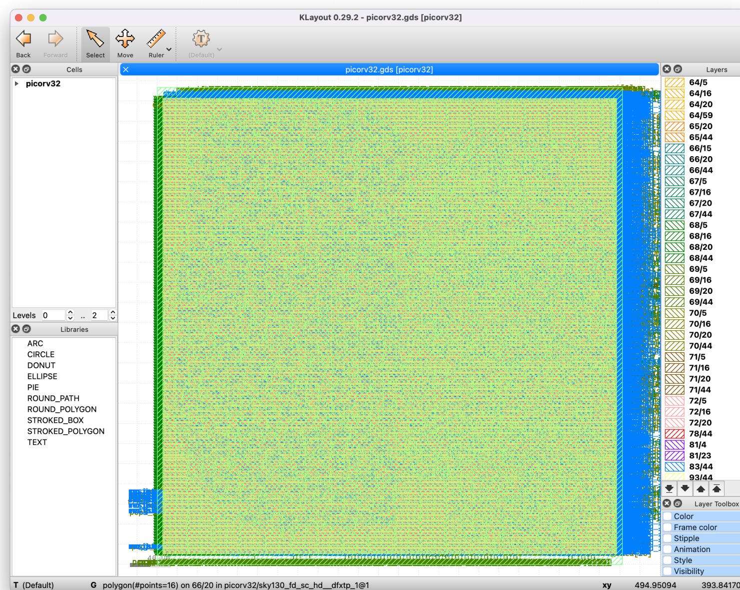

Figure 4.1: GDSII layout view of `picorv32` synthesized via OpenLane.

4.4 Case Study 2: Scratchpad Memory (SPM)

The `spm` module is a hand-written scratchpad memory, developed to explore memory timing and cell placement. It includes address decoding logic, read/write control, and synchronous registers.

Configuration

The `config.tcl` file shown in Listing 4.2 mirrors the format used for `picorv32`, with adjusted utilization targets and PDN pitch to account for the regular structure of memory arrays.

Listing 4.2: OpenLane configuration file for `spm`.

```
set ::env(DESIGN_NAME) "spm"
set ::env(VERILOG_FILES) "dir::src/*.v"
set ::env(CLOCK_PERIOD) 10
set ::env(CLOCK_PORT) "clk"
set ::env(PNR_SDC_FILE) "dir::src/spm.sdc"
set ::env(SIGNOFF_SDC_FILE) "dir::src/spm.sdc"

# PDN Configuration
set ::env(FP_PDN_VOFFSET) 5
set ::env(FP_PDN_HOFFSET) 5
set ::env(FP_PDN_VWIDTH) 2
set ::env(FP_PDN_HWIDTH) 2
set ::env(FP_PDN_VPITCH) 30
set ::env(FP_PDN_HPITCH) 30
set ::env(FP_PDN_SKIPTRIM) 1

# Pin Order
set ::env(FP_PIN_ORDER_CFG) "dir::pin_order.cfg"

# Sky130-specific overrides
set ::env(FP_CORE_UTIL) 45
set ::env(CLOCK_PERIOD) 10
set ::env(MAX_FANOUT_CONSTRAINT) 5 ;# for LS cells
```

GDSII Output

The GDS layout reveals a highly regular floorplan with dense cell packing and minimal congestion. Power stripes are clearly visible. The output file is `spm.gds`.

4.5 RTL Synthesis with Design Compiler

Although OpenLane supports full RTL-to-GDSII physical design, industry flows often decouple logic synthesis and backend layout. For RTL-level synthesis, we also evaluated Synopsys

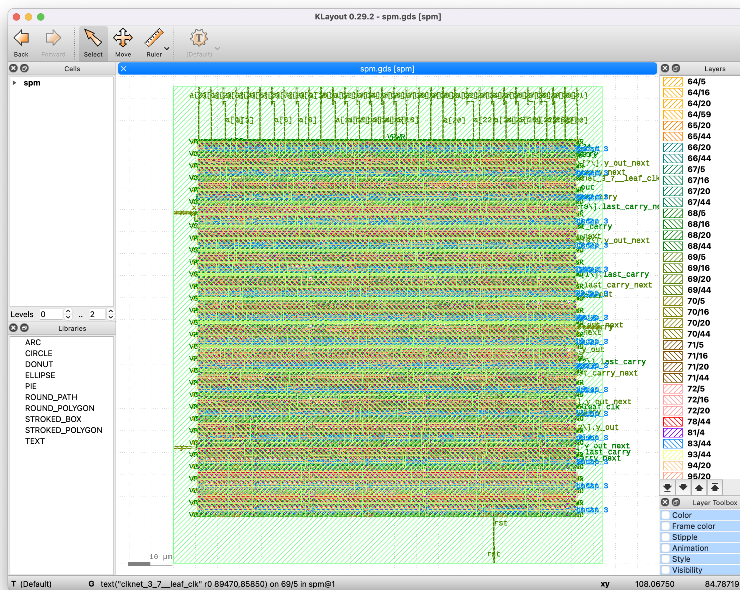

Figure 4.2: GDSII layout of `spm` memory block post-OpenLane flow.

Design Compiler (DC) on the `alex35_final` design. DC provides early feedback on logic structure, fanout, critical paths, and area breakdown, which can guide pipelining and modular refactoring.

In our DC run, we targeted a 6 ns clock period (166.66 MHz). The tool produced zero slack, indicating that the design narrowly met timing without margin. Figure 4.3 shows the setup timing summary. It includes contributions from clock uncertainty, library setup time, and data arrival time. The critical path delay closely matches the required time, confirming a well-formed sequential structure with no setup violations.

Beyond timing, Design Compiler provides a detailed report of cell counts and area utilization. The synthesized netlist contains:

- **22,461** logic cells
- **18,807** combinational gates
- **3,504** sequential flip-flops or latches
- **5,324** buffers/inverters
- Total cell area of **35,707.04** $\mu\mathbf{m}^2$

These figures are visualized in Figure 4.4, which confirms the design's moderate complexity for a compute kernel. Net interconnect area remains undefined due to zero wireload model assignment, but logic-only estimation suffices at this early stage.

```
clock clk (rise edge)                         6.00       6.00
clock network delay (ideal)                   0.00       6.00
clock uncertainty                            -0.30       5.70
outv_5_reg[3]/CK (DFF_X1)                     0.00       5.70 r
library setup time                           -0.04       5.66
data required time                                       5.66
-----------------------------------------------------------
data required time                                       5.66
data arrival time                                       -5.65
-----------------------------------------------------------
slack (MET)                                              0.00
```

Figure 4.3: Timing report from Design Compiler showing critical path delay matching the 6 ns target [18].

```
Number of ports:                              327
Number of nets:                             24111
Number of cells:                            22461
Number of combinational cells:              18807
Number of sequential cells:                  3504
Number of macros/black boxes:                   0
Number of buf/inv:                           5324
Number of references:                          38

Combinational area:                19858.762295
Buf/Inv area:                       3005.002017
Noncombinational area:             15848.279426
Macro/Black Box area:                  0.000000
Net Interconnect area:          undefined  (Wire load has zero net
area)

Total cell area:                   35707.041720
Total area:                     undefined
1
```

Figure 4.4: Area breakdown report from Design Compiler for `alex35_final` [18].

Together, these results validate the synthesizability of `alex35_final` before feeding the RTL into downstream OpenLane or ASIC toolchains. While OpenLane integrates Yosys for open-source synthesis, Design Compiler remains valuable for industry signoff or commercial PDKs.

4.6 Discussion and Summary

This chapter demonstrated how real-world RTL designs can be transformed into manufacturable layouts through a modern ASIC design flow using OpenLane and the Sky130 PDK. Both case studies—`picorv32`, a compact RISC-V processor core, and `spm`, a scratchpad memory module—highlight the accessibility and scalability of RTL-to-GDSII pipelines built on open-source EDA tools.

In each flow, a carefully configured `config.tcl` file allowed OpenLane to guide the design through approximately twenty backend stages, from synthesis and floorplanning to detailed routing and signoff DRC/LVS checks. Visual inspection of the final layouts using KLayout confirmed correct standard cell placement, track routing, and PDN grid generation. Importantly, the process required minimal manual tuning thanks to mature OpenLane defaults and clean Verilog sources.

While commercial tools such as Synopsys Design Compiler and IC Compiler II (ICC2) provide finer control over cell selection, timing budgeting, and power management, the open-source stack is increasingly viable for academic research and teaching. Design Compiler was additionally used to evaluate the `alex35_final` accelerator, producing detailed area and timing reports that inform future pipelining decisions. These insights are particularly useful when refining datapath-heavy blocks.

Looking forward, future extensions of this work could include:

- Exploring custom constraints for tighter timing closure.
- Automating power-domain-aware synthesis and floorplanning.
- Integrating formal verification steps within the RTL-to-GDSII loop.
- Performing back-annotated simulations with extracted parasitics (SPEF).

Together, these experiments establish a robust and transparent workflow for ASIC design from RTL to layout, using both open and commercial EDA tools. The resulting methodology serves as a foundation for building more complex SoC components in subsequent work.

Chapter 5

Machine Learning for Hardware

The intersection of machine learning (ML) and electronic design automation (EDA) has become a rich research area. As digital integrated circuit (IC) designs grow in complexity and size, traditional rule-based or heuristic-driven optimization techniques face limitations in scalability and adaptability. ML offers the promise of data-driven modeling, prediction, and guidance, making it increasingly essential in modern hardware design flows.

Recent surveys such as that by Cong et al. [3] highlight the transformative role of ML in high-level synthesis (HLS), where predictive models and reinforcement learning have improved result estimation and design-space exploration. More broadly, Huang et al. [4] categorize ML contributions to EDA into four categories: performance prediction, cross-platform generalization, active learning, and enhancement of conventional algorithms. Figure 5.1 reproduces a summary of ML models used across these subareas in HLS flows. These range from tree-based models (e.g., XGBoost, Random Forests) to deep learning (e.g., ANN, GNN), applied across tasks such as throughput estimation, platform retargeting, or Pareto optimization.

Machine learning's role in hardware spans beyond HLS. For logic synthesis, ML has been employed to predict timing violations and guide retiming. In physical design, placement and routing tools have begun to incorporate learned congestion estimators. In verification, ML models help identify corner-case scenarios, suggest high-coverage test stimuli, and prioritize simulation paths based on historical waveform patterns. Even hardware security, such as detecting trojans or identifying information leakage, benefits from ML-based classification and anomaly detection. As toolchains become increasingly automated and data-rich, ML will likely become a built-in layer across the entire EDA stack.

5.1 Related Work: Pyramid Framework

Among existing ML-HLS frameworks, the Pyramid framework [6] presents a notable architecture that integrates feature extraction, timing/resource estimation, and Pareto-front

Section	Task	ML Algorithm
Result prediction	Timing and resource usage prediction	Lasso, ANN, XGBoost
	Max frequency, throughput, area	Ridge regression, ANN, SVM, Random Forest
	Latency	Gaussian Process
	Operation delay	Graph Neural Network
Cross-platform	Predict for new FPGA platforms	ANN
	Predict for new applications through executing on CPUs	Linear models, ANN, Random Forest
Active learning	Reduce prediction error with fewer samples	Random Forest, Gaussian Process Regression
	Reduce prediction error for points near the Pareto-frontier	Gaussian Process
	Reduce the risk of losing Pareto designs	Random Forest
Improving conventional algorithms	Initial point selection	Quadratic regression
	Generation of new sample	Decision Tree
	Hyper-parameter selection	Decision Tree

Figure 5.1: Representative ML applications in HLS design tasks, adapted from [4].

learning. The flow extracts features from HLS designs at both the IR and RTL levels, trains ensemble regressors to predict area and delay, and then uses a Gaussian Process model to identify optimal tradeoffs. Figure 5.2 outlines the key stages in Pyramid.

This framework highlights the effectiveness of combining symbolic features (e.g., loop nesting depth, array partitioning) with empirical metrics (e.g., synthesis area) to build accurate predictors. It also demonstrates how ML workflows can be structured to provide actionable guidance in design space exploration rather than only after-the-fact evaluation. While Pyramid is highly extensible, it focuses primarily on inference from fixed design datasets. In contrast, our case study below constructs a full toolchain that integrates feature selection, regression, evaluation, and visualization for both design and prediction.

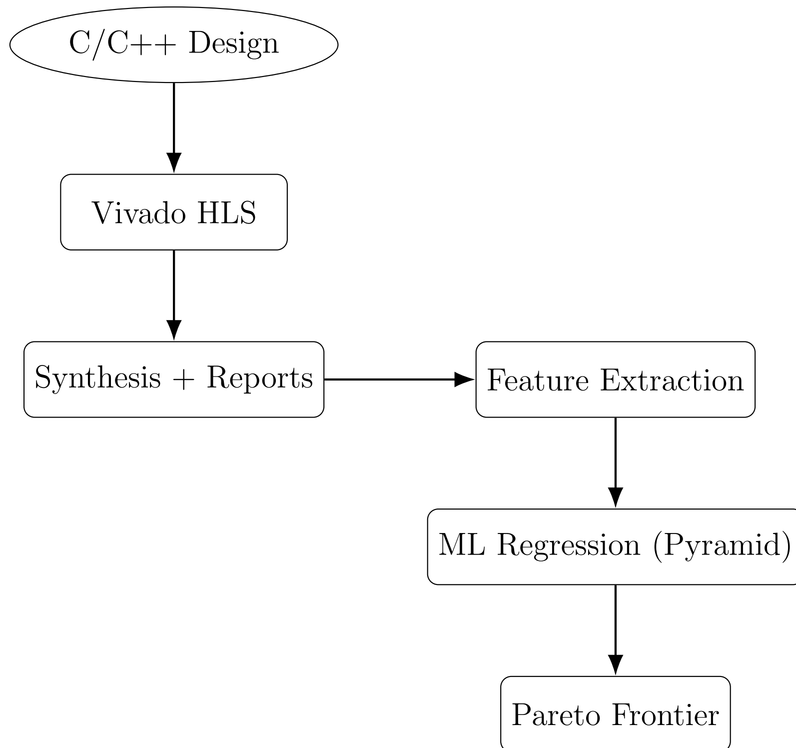

Figure 5.2: Reconstructed flow of Pyramid: HLS feature extraction and ML-driven design prediction [6].

5.2 Case Study: ML-Guided Timing Deviation Prediction

High-Level Synthesis (HLS) tools have significantly accelerated the hardware design process by allowing developers to describe functionality in high-level languages such as C/C++ or SystemC and automatically generate RTL. Despite their promise, HLS-generated designs often exhibit discrepancies between estimated and actual timing, leading to suboptimal resource usage or timing violations. Accurately predicting these deviations remains a challenging but crucial task in design closure.

To evaluate the effectiveness of machine learning in this context, we conducted a case study using the open-source *HLSdataset* [7], which contains 14,487 synthesized designs covering a variety of compute kernels and optimization conditions. Each entry includes post-synthesis resource and timing data, such as flip-flop (FF), lookup table (LUT), and DSP usage—along with both target and estimated clock periods. Our goal was to model the deviation between estimated and target clock periods, which reflects how far off the tool's internal predictions are from the user's performance constraints.

The pipeline begins with extracting features from Vivado HLS synthesis reports. These

features include counts of arithmetic operations (`c_num_arith`), logic operations (`c_num_logic`), DSP usage, and other synthesized resource values. While these metrics are derived post-synthesis, they encode rich information about control and datapath complexity, making them suitable inputs for regression models.

We trained a Random Forest regression model to predict the deviation between the target and estimated clock periods. The model was evaluated using mean absolute error (MAE) and the coefficient of determination (R^2). Our best-performing model achieved an MAE of 1.084 ns and an R^2 score of 0.448, suggesting moderate success in approximating timing deviations without full re-synthesis.

To enhance interpretability, SHAP (SHapley Additive exPlanations) [8] analysis was applied to the trained model. Figure 5.3 shows that the most influential features are the number of arithmetic and logic operations—consistent with the hypothesis that more complex datapaths tend to increase critical path length, reducing timing slack. This validates the model's alignment with intuitive hardware design principles.

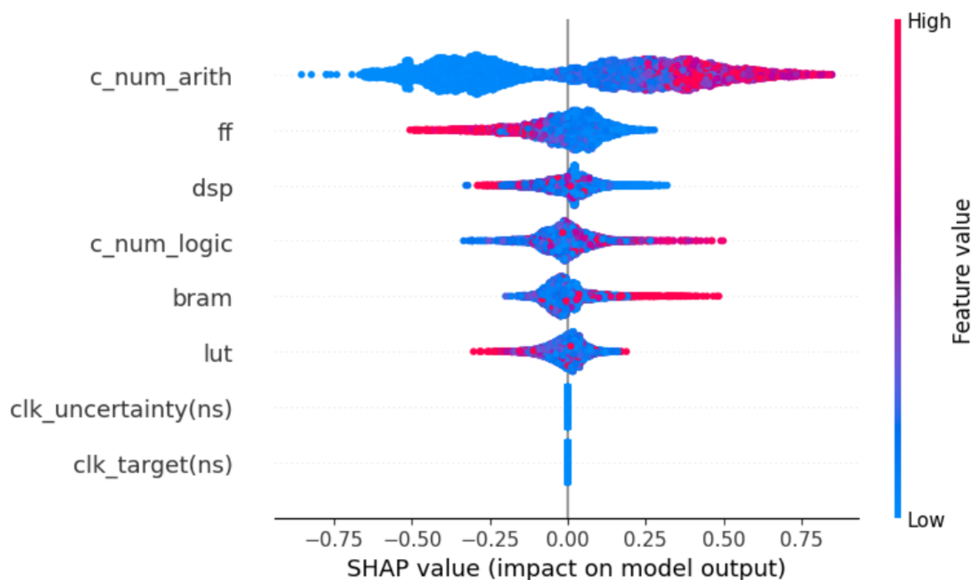

Figure 5.3: SHAP Analysis for Feature Importance

This case study confirms that tree-based models like Random Forest can serve as lightweight predictors of HLS timing deviation, providing early feedback on whether a design is likely to meet timing. While the accuracy is bounded by the dataset's reliance on post-synthesis features, the framework opens the door for future enhancements such as pre-synthesis modeling, incorporation of structural features from source code, and use of active learning to minimize retraining overhead. In industry settings where design space exploration is constrained by tool runtime, such ML-assisted predictions can significantly accelerate design iteration and refinement.

5.3 Summary

In this chapter, we explored how machine learning is increasingly shaping hardware design flows, with a particular focus on high-level synthesis (HLS). We reviewed key literature highlighting ML's applications in performance estimation, cross-platform prediction, and design optimization. The Pyramid framework served as a concrete example of ML-enhanced design flow, offering insights into model-guided design space exploration.

To ground these ideas, we presented a case study using Random Forest regression to predict timing deviation in Vivado HLS designs. By analyzing SHAP values, we gained interpretable insights into which synthesis features most strongly influence timing outcomes. This confirms that even with relatively simple ML models and standard feature sets, practical and actionable predictions can be obtained.

These results demonstrate the growing feasibility of ML-guided HLS workflows, setting the stage for more sophisticated applications in broader design contexts. In the next chapter, we extend these concepts to ASIC and explore the role of ML and large language models (LLMs) in ASIC design automation.

Chapter 6

LLM-Assisted ASIC Design

Large Language Models (LLMs) such as GPT-4, Claude, and Gemini have recently made significant advances in hardware design automation. While LLMs were initially applied to natural language and software tasks, their ability to parse and generate structured code makes them well-suited for hardware description languages (HDLs) like Verilog and VHDL. The availability of open-source repositories, modular design examples, and standardized toolchains further enhances the feasibility of LLM-guided ASIC development.

This chapter surveys the emerging roles of LLMs across the ASIC design pipeline, including RTL generation, analog circuit synthesis, design verification, specification extraction, and EDA tool assistance. Rather than focusing solely on one subdomain such as in-memory computing (IMC), we provide a broader perspective on how transformer-based models are enabling new levels of automation, code generation, and tool integration in digital and analog hardware design.

6.1 Emerging Roles of LLMs in ASIC Design

Recent research efforts have demonstrated that LLMs can automate various parts of the ASIC workflow:

- **RTL and Circuit Generation**: Transforming high-level intent into Verilog, VHDL, or SPICE netlists.
- **Tool Scripting and EDA Assistance**: Generating synthesis and simulation scripts, or interpreting tool outputs.
- **Verification and Assertion Synthesis**: Writing SystemVerilog Assertions (SVA), testbenches, or coverage models.
- **Specification Extraction**: Converting design specifications in natural language to HDL-style module templates.

This expansion has been enabled by a combination of prompt engineering, code+HDL pretraining, and retrieval-augmented generation (RAG), which help models better interpret structural design patterns. Architectural refinements such as multi-agent collaboration and fine-tuning with formal tools also improve the reliability of generated artifacts. Table 6.1 summarizes representative frameworks that apply LLMs across digital, analog, and EDA-assistive workflows, capturing their target domain, input modalities, and output formats.

Table 6.1: Representative LLM-Based Frameworks for ASIC Design and Verification

Framework	Domain	Output Format	User Input
ChipGPT [19]	RTL Generation (Digital)	Verilog RTL	Specification + goals
LIMCA [20]	Analog IMC Design	SPICE Netlist	Performance targets
AssertLLM [21]	Verification	SystemVerilog Assertions	Bug descriptions / signals
ChatEDA [22]	EDA Assistance	Tool Scripts (e.g., Yosys, OpenLane)	RTL + design goal
SpecLLM [23]	Spec Extraction	HDL-like Spec Drafts	Natural language design intent
UVLLM [24]	UVM Debug Automation	Corrected RTL (SystemVerilog)	RTL + faulty testbench
AnalogXpert [25]	Analog Design	Modular SPICE Netlist	Architecture + constraints
HDLGen [26]	Mixed RTL/Testbench Gen	Verilog/VHDL + TB	Design goal + project type
LLM4DV [11]	Digital Verification	Verilog Testbenches	Functional spec + RTL intent
Masala-CHAI [27]	Schematic Translation	SPICE Netlist	Circuit schematic image
TPU-Gen [28]	Accelerator Design	Verilog (systolic TPU)	High-level array spec
ChipNeMo [29]	EDA Report Generation	Scripts / Summaries	Tool logs or prompts
C2HLSC [30]	HLS Compilation	HLS C Code	C source code + streaming pattern
AutoChip [31]	RTL Generation (Digital)	Verilog RTL	Natural-language module specification
RTLLM [32]	RTL Generation (Digital)	Verilog RTL	Natural-language design instructions
SPICEPilot [33]	Analog Design	SPICE Netlist + Sim. Script	Circuit description + analysis goal
LLMCompass [34]	EDA Evaluation	Arch. performance model	Hardware parameters + LLM model

6.2 Representative Case Studies

To illustrate the diversity and impact of LLM-based approaches in ASIC design, we summarize five representative works, each focusing on a different aspect of the workflow.

ChipGPT: Digital RTL Generation from Specifications

ChipGPT [19] focuses on full-stack RTL generation from natural-language descriptions. Users provide high-level specifications such as "8-bit ALU with add, sub, and AND" and the tool produces synthesizable Verilog modules, often with basic testbenches.

ChipGPT combines prompt engineering with architectural fine-tuning to align the model's generation with hardware conventions (e.g., FSMs, bit slicing, input/output declarations). It achieves high functional correctness across microbenchmark modules and introduces correction loops that refine output based on simulation feedback, bridging the gap between raw LLM output and industrial RTL standards.

LIMCA: Analog In-Memory Computing Circuit Synthesis

LIMCA [20] tackles a harder problem: generating analog SPICE netlists for in-memory computing (IMC) circuits from performance constraints such as "10-bit dot-product at 100 MHz."

It combines prompt-guided synthesis with reinforcement learning to adapt SPICE topologies to constraints like power, speed, and signal-to-noise ratio. LIMCA also incorporates a simulation validation loop—each LLM-generated netlist is tested via Ngspice, and failed generations are used as negative samples in fine-tuning. The tool achieves up to 80% usable SPICE generation for SRAM-based and capacitor-based IMC macros, and enables rapid architecture co-exploration for analog computing blocks.

ChipNeMo: EDA Command and Task Assistance

ChipNeMo [29] demonstrates how LLMs can enhance user interaction with EDA tools. Rather than producing HDL code, it translates user intentions (e.g., "run synthesis with higher clock margin") into valid tool commands for environments like Synopsys Design Compiler or Cadence Innovus.

ChipNeMo supports natural-language querying, auto-completion of TCL scripts, and suggestion of tool flags based on context. It's powered by a retrieval-augmented backend and a domain-adapted instruction-tuned LLM. Although it doesn't generate RTL, its utility lies in bridging human-tool interaction, especially for novices or rapid design iteration.

AssertLLM: Assertion Generation for Formal and Simulation Use

AssertLLM [21] targets a critical verification task: writing SystemVerilog Assertions (SVA) from textual bug specifications or protocol descriptions. Given inputs like "If 'go' is high, 'done' must be high within 5 cycles," it generates SVAs such as:

Listing 6.1: Assertion generated by AssertLLM

```
property p_go_done;
  @(posedge clk) go |-> ##[1:5] done;
endproperty
assert property(p_go_done);
```

The model is trained on a paired corpus of English-SVA examples and integrates syntax-aware constraints during decoding. AssertLLM significantly reduces time to write and validate assertions, and can be integrated with simulation frameworks like VCS or formal engines like VC Formal.

SpecLLM: HDL Specification Drafting

SpecLLM [23] addresses the problem of drafting HDL documentation or design intent from rough textual descriptions. It converts natural language ("SPI controller with configurable

clock polarity and phase") into structured interface descriptions, port listings, and behavioral outlines.

This tool is useful in early design stages or for onboarding engineers. Its output is not yet full RTL, but resembles 'spec' files used in some commercial flows. SpecLLM offers a light-weight alternative to manual specification documents, supporting both Englishspec and specVerilog pathways.

6.3 Discussion and Outlook

These examples demonstrate that Large Language Models (LLMs) are increasingly capable of contributing to nearly every phase of the ASIC design workflow: from early-stage specification drafting to RTL generation, analog circuit synthesis, verification automation, and EDA tool assistance. This breadth of capability is unprecedented and suggests a paradigm shift in how hardware design may be conducted in the near future.

Each design domain presents different strengths and limitations for LLM-based tools:

- **Digital RTL generation** has become feasible with moderate success. LLMs can produce functional Verilog for microbenchmark-scale designs, but correctness verification through simulation or formal tools remains essential. Multi-module interaction, state encoding, and timing closure are often missed in first-pass generations.

- **Analog circuit generation**, particularly SPICE-level synthesis, remains highly challenging. However, with simulation-in-the-loop prompting (as used in LIMCA and SPICEPilot), LLMs can converge on usable topologies. Analog design also suffers less from syntax errors but is more sensitive to small circuit-level variations, requiring tighter validation.

- **Verification and assertion synthesis** tools like AssertLLM and LLM4DV show promise in accelerating tedious manual tasks. These tools excel when trained with domain-specific corpora, but still require user oversight to catch vacuous conditions or false positives in coverage metrics.

- **EDA assistance** tools such as ChipNeMo and LLMCompass make the user-tool interaction more accessible, especially for early-career engineers. They help interpret error messages, generate TCL scripts, or map informal goals to tool-compatible constraints.

Despite their promise, LLMs in ASIC design face several challenges:

- **Hallucination and Incomplete Specifications**: LLMs often make confident but incorrect assumptions in designs, especially when key constraints are missing from the prompt. This leads to syntactically correct but functionally flawed RTL.

- **Multi-Module and Interface Coherence**: While LLMs can generate standalone modules, ensuring consistency across interacting modules (e.g., handshake protocols, shared memory interfaces) remains an open issue.
- **Verification Gaps and Safety Hazards**: Automatically generated assertions may be vacuous or unsound. Without formal coverage analysis, designers risk over-trusting the output.
- **Data Scarcity and Domain Transfer**: There is a lack of high-quality, labeled datasets for analog circuits, formal specs, or post-layout EDA tasks. This limits the effectiveness of supervised fine-tuning and hampers generalization to novel architectures or IP.

To address these challenges, recent work highlights several promising **future directions**:

- **Closed-Loop Design-Verify Cycles**: Embedding the LLM within a feedback loop that includes simulation, formal checking, or physical synthesis can help correct hallucinations and improve the realism of generated artifacts.
- **Multi-Agent Collaboration**: Inspired by human design teams, multiple specialized agents (e.g., spec writer, RTL generator, testbench synthesizer, formal verifier) can be coordinated to communicate and refine outputs, potentially outperforming monolithic prompting.
- **Hardware-Aware Pretraining and Fine-Tuning**: Most current LLMs are pre-trained on general code/text. Domain-adapted finetuning using HDL corpora, simulation logs, and waveform data could yield higher-quality outputs and better contextual understanding.
- **Design-Compiler Integration**: Bridging the LLM with compiler passes from tools like Vivado HLS or OpenROAD enables real-time performance feedback and tighter alignment between generated code and backend constraints.
- **Interactive Co-Design Interfaces**: By embedding LLMs in IDEs or HLS GUIs, users can receive contextual prompts, warnings, or optimization hints—enabling tighter synergy between human designers and machine-generated suggestions.

In summary, LLM-assisted ASIC design is no longer a speculative idea, it is an active research frontier with functioning tools across multiple domains. While significant hurdles remain, particularly in trustworthiness and complex integration, the trajectory is clear: generative models will play a major role in automating, accelerating, and augmenting the future of hardware design.

In the next chapter, we reflect on how these trends shape the future of hardware education and design methodology, and how they intersect with efforts in hardware security and formal verification.

Chapter 7

Hardware Security

7.1 Overview: Security Verification in Practice

Modern System-on-Chip (SoC) designs are increasingly exposed to sophisticated threats such as hardware Trojans, protocol violations, information leakage, and side-channel vulnerabilities. Traditional functional verification techniques are insufficient to uncover these issues, especially when security relies on correct FSM behavior, reset handling, and temporal properties. In this context, security verification blends formal methods, simulation, taint tracking, and structured code analysis.

To gain practical experience in this domain, I participated in the **Hack@DAC 2025 International Hardware Security Competition**. This annual competition simulates industry-scale security auditing, requiring participants to identify, exploit, and patch real bugs in RTL-level SoC designs. Through this challenge, I applied simulation, formal property checking, static linting, taint tracking, and even LLM-assisted tooling to develop a robust security assessment pipeline.

7.2 Hack@DAC: Competition Background

Hack@DAC is hosted at the ACM/IEEE Design Automation Conference, with two highly demanding phases:

- **Phase 1:** Teams analyze RTL designs locally to identify vulnerabilities and submit security bug reports with mitigation strategies and classification.
- **Phase 2:** Finalists are given remote access to a full SoC and must automate the exploitability workflow, demonstrate working attacks, and propose LLM-assisted tooling.

In 2025, the competition focused on **OpenTitan**, a secure open-source RISC-V SoC that includes UART, SRAM, I2C, and system-level alerting subsystems. Our team placed **third**,

with a successful demonstration of three confirmed security bugs, tool-based detection, and automation pipelines based on formal and LLM techniques.

7.3 Toolchain Overview: Synopsys + Custom Automation

Our analysis was powered by a combination of industry-standard Synopsys tools and custom-developed automation:

VCS Simulation and Verdi Debugging

We used **VCS** to simulate functional behavior and inject malicious stimuli. Combined with **Verdi**, we traced signal propagation, FSM transitions, and glitch-based side channels across modules like I2C and SRAM. Verdi's FSM viewer and backward trace feature were essential for debugging premature transitions and timing bugs.

Figure 7.1 illustrates a simulation session in Verdi where an assertion violation `AcqDepthRdCheck_A` is flagged during the I2C FSM state transition. The waveform reveals a critical timing mismatch: the FSM transitions from `Setup` to `TransmitPulse` even when the FIFO depth signal is insufficient, which can only be observed via wave-level inspection. This demonstrates how Verdi's deep trace capabilities are vital for diagnosing assertion failures and understanding internal behavior.

VC Formal: Assertion Checking and Equivalence

VC Formal was used in two major modes:

- **FPV (Formal Property Verification):** Ran `prim_assert.sv` assertions to catch violations like unexpected STOP conditions in I2C.
- **FSV (Formal Sequential Verification):** Compared pre- and post-edit RTLs for equivalence and Trojan detection.

We wrote additional assertions for reset behavior, arbitration control, and secret erasure, which VC Formal flagged as violated under adversarial sequences.

Figure 7.2 shows the VC Formal dashboard with results from FPV analysis. Assertions like `SclOutputGlitch_A` and `AcqDepthRdCheck_A` failed at different depths, indicating violations such as clock glitch propagation and illegal FIFO reads. This GUI also provides vacuity analysis and witness traces, which were crucial in localizing the faults.

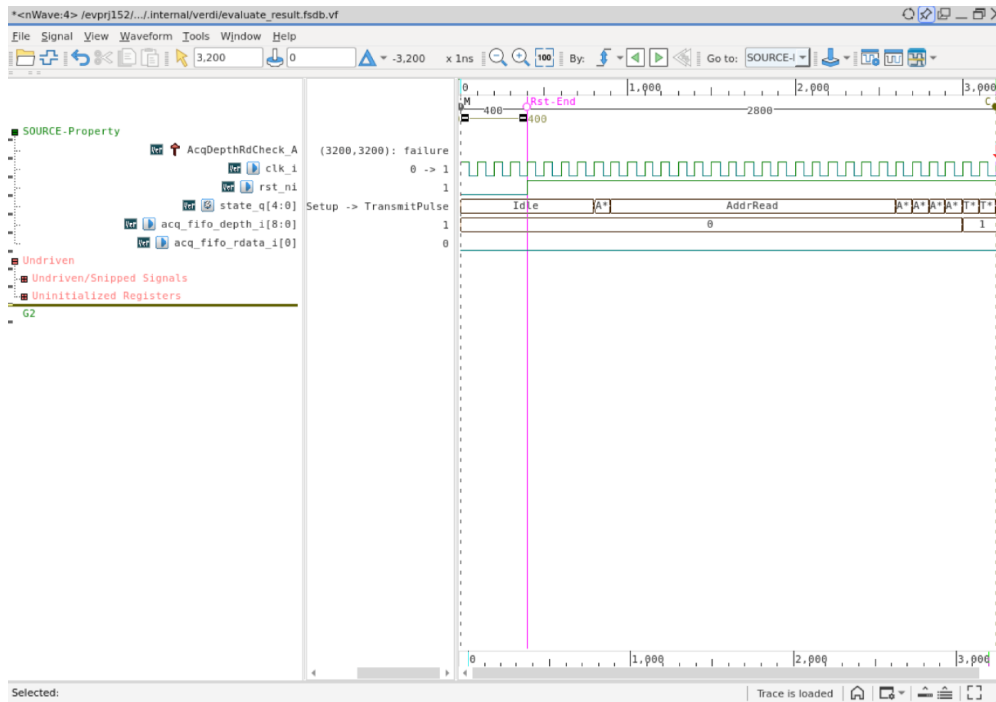

Figure 7.1: VCS and Verdi used to simulate OpenTitan IPs and trace internal FSM signals [15].

SpyGlass Lint

SpyGlass Lint enabled pre-synthesis code analysis. We flagged:

- Uninitialized registers
- Incomplete reset logic
- Unsafe FSM transitions

These helped pre-identify modules with potential leakage or logic confusion vulnerabilities.

Figure 7.3 displays the SpyGlass Lint output from the AES controller design. The error (W110) highlights a 32-bit port width mismatch in the instantiation of the shadow register module. This type of issue, if undetected, could propagate silently to downstream logic and cause unpredictable behavior in hardware, especially in security-critical paths.

TPROP: Taint Tracking and Alert Path Validation

We used Synopsys TPROP in conjunction with formal assertions to trace sensitive control paths such as alert lines, encryption status, and secure FSM transitions. While full graphical

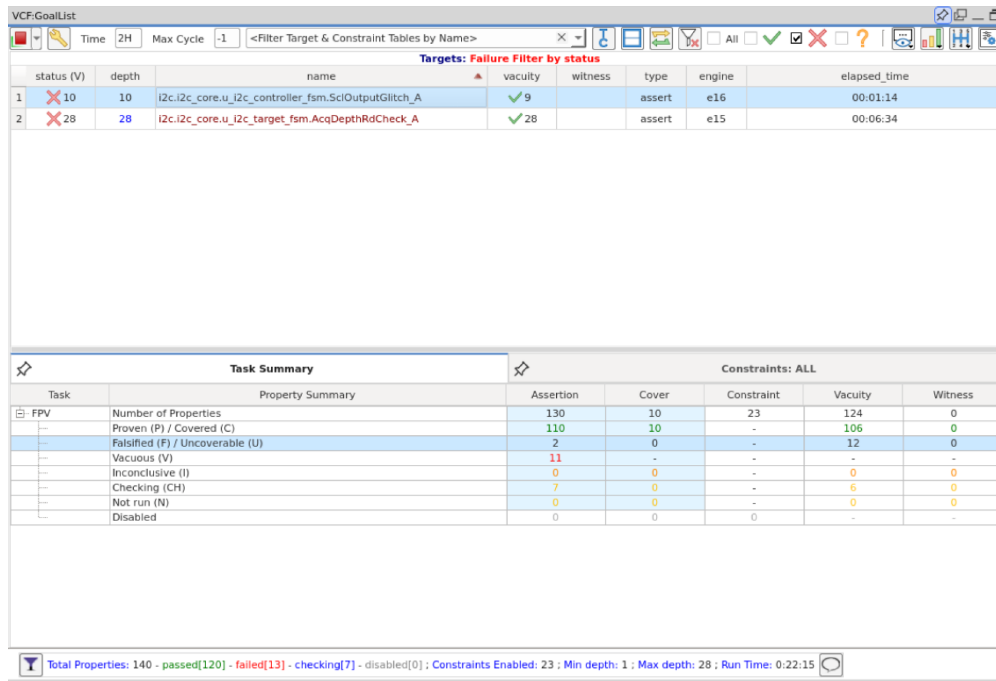

Figure 7.2: VC Formal used for assertion coverage and sequential equivalence checking [35].

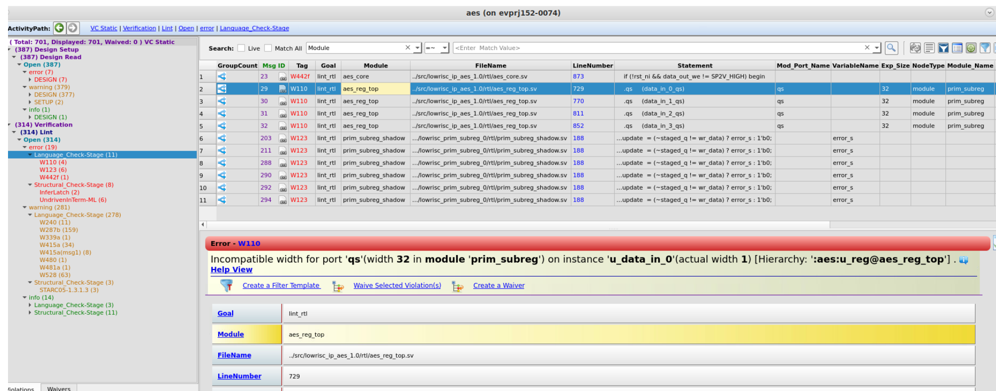

Figure 7.3: SpyGlass Lint output used to catch coding flaws and reset misbehavior [36].

taint propagation views were not exported, TPROP was integrated into our assertion-driven flow to detect propagation failures across alert channels.

As shown in Figure 7.4, a taint-related assertion (`AlertPKnown0_A`) failed during simulation at 762,785 ps. This demonstrates how insecure or uncontrolled alert outputs can be flagged using property checks under taint initialization and propagation. The signal `alert_tx_o` remains undefined or misaligned with security policy, pointing to a critical path that requires redesign or additional gating logic.

Figure 7.4: TPROP-assisted assertion failure highlighting propagation flaw in `alert_tx_o` output [37].

7.4 Automated Detection Workflow

To accelerate bug discovery and reduce manual effort, we designed three lightweight but powerful tools:

Tool 1: Static Code Augmenter

Custom rule-based static checker to enhance FSM extraction and analyze improper transitions between security states (e.g., jumping from BUSY to IDLE while skipping ALERT).

Tool 2: Python Script for Bug Category Detection

This script parsed Verilog and ASTs to identify missing reset initializations, register write races, and unguarded alert triggers. It helped us triage bug candidates across multiple IP blocks.

Tool 3: LLM-Based Prompt-Driven Testbench Generator

We used a YAML-based pipeline where prompts like:

> *"Generate a testbench to toggle STOP and START in the middle of a transaction while observing alert behavior."*

were fed into an LLM to produce customized SystemVerilog testbenches with assertions and stimulus. The outputs were iteratively refined and simulated using VCS.

7.5 Case Studies: Three Exploit Examples

Bug 1: Alert Suppression via Premature IDLE

Description: In the I2C FSM, an error state transitioned to IDLE before alert logic could activate, allowing silent protocol violations.

Pseudocode:

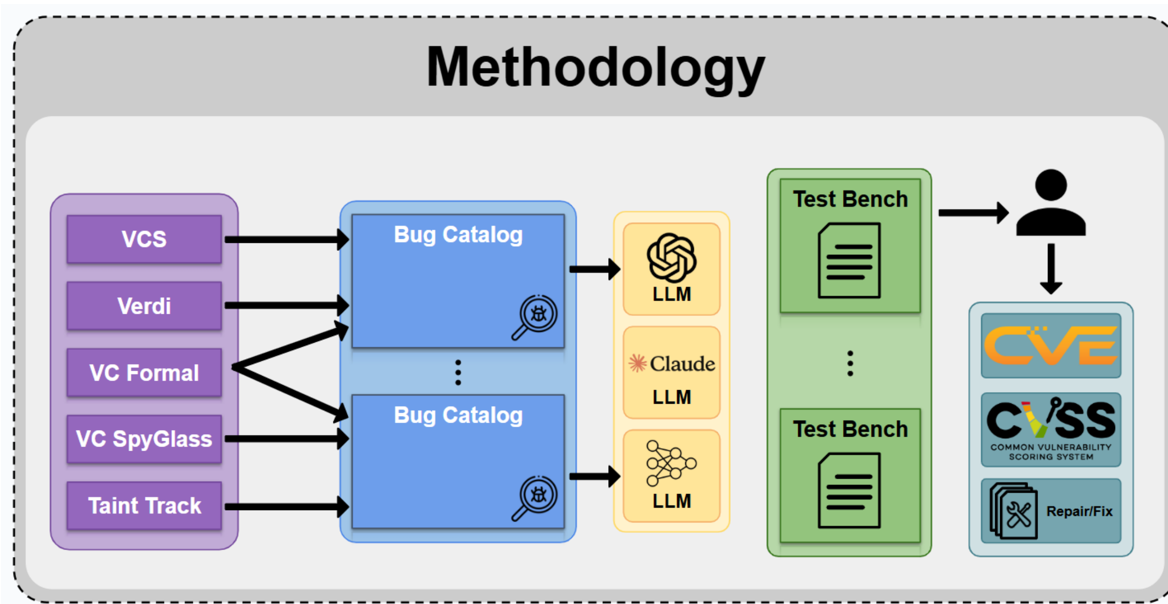

Figure 7.5: Custom LLM and script-based workflow for automated test generation.

```
if (fsm_state == ERROR && !alert_sent)
    fsm_state <= IDLE; // suppresses error!
```

Impact: Undermines alert propagation, allowing stealthy violations.
Mitigation: Block IDLE state entry unless all pending alerts are issued.

Bug 2: Uninitialized Secure Access Control

Description: A secure SRAM access flag was not reset properly, and its default value allowed early memory access.

Pseudocode:

```
logic allow_secure_read;
if (allow_secure_read)
    read_key();
```

Impact: Risk of unprotected SRAM access before lifecycle or key programming.
Mitigation: Add synchronous reset for all security-critical state registers.

Bug 3: Glitch on SDA Causes Arbitration Loss

Description: The SDA signal glitched for one cycle during an ACK phase, but the FSM didn't detect the violation due to lack of filtering.

Pseudocode:

```
assign sda_o = glitch ? 0 : ack_value;
```

Impact: Causes arbitration loss or silent ACK mismatches on I2C.
Mitigation: Add input deglitch filter or two-stage synchronizer on SDA.

7.6 Summary and Reflections

This project showcased the need for rigorous hardware security testing that goes beyond typical verification flows. We learned that:

- Formal tools (e.g., VC Formal) can catch deep temporal bugs.
- Lint and taint tools flag structural vulnerabilities early.
- LLMs offer scalable testbench and assertion synthesis when guided by domain-specific prompts.

Hack@DAC not only tested our verification skills but also forced us to think like attackers—anticipating failure modes and reasoning about secure-by-construction design. This hybrid approach, combining commercial tools, scripting, and LLM pipelines—demonstrates the future of security-aware hardware verification.

Chapter 8

Conclusion and Future Work

This thesis presented an integrated perspective on modern ASIC design and verification, bridging traditional RTL workflows with emerging techniques in machine learning and language model automation. Beginning with core design blocks—FSMs, matrix multiplication, neural network accelerators, and memory controllers. we built a strong RTL foundation and demonstrated complete synthesis flows using OpenLane and the Sky130 process node.

Through detailed toolchain walk-throughs and experiments, we showed how verification with Synopsys VCS, Verdi, and VC Formal exposes functional bugs and security flaws, especially when combined with assertions and waveform tracing. Our Hack@DAC 2025 participation applied this methodology in practice, where we uncovered and analyzed vulnerabilities in OpenTitan using commercial tools and custom scripts. Three representative bugs were dissected, highlighting root causes such as improper FSM gating, reset handling, and glitch sensitivity.

The thesis also contributes to the expanding field of ML and LLM-assisted hardware automation. We trained a Random Forest model on HLS data to predict timing/resource metrics and surveyed more than 20 frameworks utilizing LLMs for HDL generation, analog design, spec drafting, and assertion synthesis. The case studies of ChatEDA, ChipGPT, LIMCA, and AssertLLM illustrated the diversity and maturity of LLM-based design assistance.

Looking ahead, we see strong opportunities in:

- Closed-loop RTL generation and formal verification via multi-agent LLMs
- Integrating waveform traces, timing reports, and coverage feedback into LLM prompts
- Automating security vulnerability discovery using taint analysis and generative testbenches

By combining domain knowledge with the generative reasoning of LLM, hardware design can become faster, more scalable, and inherently safer. This thesis sets the groundwork for such innovation and serves as a reference for future work on AI-assisted EDA and secure hardware systems.

Bibliography

[1] The OpenROAD Project. *OpenLane User Guide*. 2024. URL: https://github.com/The-OpenROAD-Project/OpenLane.

[2] Shekhar Borkar and Andrew A. Chien. "The future of microprocessors". In: *Communications of the ACM* 54.5 (2011), pp. 67–77.

[3] Jason Cong et al. "FPGA HLS today: successes, challenges, and opportunities". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 15.4 (2022), pp. 1–42.

[4] Guyue Huang et al. "Machine learning for electronic design automation: A survey". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 26.5 (2021), pp. 1–46.

[5] Jason Cong et al. "Machine learning for electronic design automation: A survey". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 26.5 (2021), pp. 1–46.

[6] Hosein Mohammadi Makrani et al. "Pyramid: Machine learning framework to estimate the optimal timing and resource usage of a high-level synthesis design". In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2019, pp. 397–403.

[7] Zhigang Wei et al. "Hlsdataset: Open-source dataset for ml-assisted fpga design using high level synthesis". In: *2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. 2023, pp. 197–204.

[8] Scott M Lundberg and Su-In Lee. "A unified approach to interpreting model predictions". In: *Advances in neural information processing systems* 30 (2017).

[9] Ruidi Qiu et al. "Autobench: Automatic testbench generation and evaluation using llms for hdl design". In: *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*. 2024, pp. 1–10.

[10] Ruidi Qiu et al. "Correctbench: Automatic testbench generation with functional self-correction using llms for hdl design". In: *2025 Design, Automation & Test in Europe Conference (DATE)*. IEEE. 2025, pp. 1–7.

[11] Zixi Zhang et al. "Llm4dv: Using large language models for hardware test stimuli generation". In: *arXiv preprint arXiv:2310.04535* (2023).

[12] Heng Ping et al. "HDLCoRe: A Training-Free Framework for Mitigating Hallucinations in LLM-Generated HDL". In: *arXiv preprint arXiv:2503.16528* (2025).

[13] Synopsys Inc. *Synopsys VCS User Guide*. Version W-2024.09-1. 2024. URL: https://training.synopsys.com/learn/courses/290/vcs-rtl-and-gate-level-simulation/lessons.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012).

[15] Synopsys. *Synopsys Verdi*. 2025. URL: https://www.synopsys.com/verification/debug/verdi.html.

[16] SkyWater Technology. *SkyWater SKY130 PDK*. 2025. URL: https://github.com/google/skywater-pdk.

[17] YosysHQ. *PicoRV32 - A Size-Optimized RISC-V CPU*. 2025. URL: https://github.com/YosysHQ/picorv32.

[18] Synopsys. *Synopsys Design Compiler*. 2025. URL: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.

[19] Kaiyan Chang et al. "Chipgpt: How far are we from natural language hardware design". In: *arXiv preprint arXiv:2305.14019* (2023).

[20] Deepak Vungarala et al. "LIMCA: LLM for Automating Analog In-Memory Computing Architecture Design Exploration". In: *arXiv preprint arXiv:2503.13301* (2025).

[21] Zhiyuan Yan et al. "AssertLLM: Generating Hardware Verification Assertions from Design Specifications via Multi-LLMs". In: *Proceedings of the 30th Asia and South Pacific Design Automation Conference*. 2025, pp. 614–621.

[22] Haoyuan Wu et al. "Chateda: A large language model powered autonomous agent for eda". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024).

[23] Mengming Li et al. "Specllm: Exploring generation and review of vlsi design specification with large language model". In: *arXiv preprint arXiv:2401.13266* (2024).

[24] Yuchen Hu et al. "UVLLM: An Automated Universal RTL Verification Framework using LLMs". In: *arXiv preprint arXiv:2411.16238* (2024).

[25] Haoyi Zhang et al. "AnalogXpert: Automating Analog Topology Synthesis by Incorporating Circuit Design Expertise into Large Language Models". In: *arXiv preprint arXiv:2412.19824* (2024).

[26] Fearghal Morgan et al. "HDLGen-ChatGPT Case Study: RISC-V Processor VHDL and Verilog Model-Testbench and EDA Project Generation". In: *Proceedings of the 34th International Workshop on Rapid System Prototyping*. 2023, pp. 1–7.

[27] Jitendra Bhandari et al. "Masala-CHAI: A Large-Scale SPICE Netlist Dataset for Analog Circuits by Harnessing AI". In: *arXiv preprint arXiv:2411.14299* (2024). URL: https://arxiv.org/abs/2411.14299.

[28] Deepak Vungarala et al. "TPU-Gen: LLM-Driven Custom Tensor Processing Unit Generator". In: *arXiv preprint arXiv:2503.05951* (2025).

[29] Mingjie Liu et al. "Chipnemo: Domain-adapted llms for chip design". In: *arXiv preprint arXiv:2311.00176* (2023).

[30] Luca Collini, Siddharth Garg, and Ramesh Karri. "C2HLSC: Leveraging Large Language Models to Bridge the Software-to-Hardware Design Gap". In: *ACM Transactions on Design Automation of Electronic Systems* (2024).

[31] Shailja Thakur et al. "Autochip: Automating hdl generation using llm feedback". In: *arXiv preprint arXiv:2311.04887* (2023).

[32] Yao Lu et al. "Rtllm: An open-source benchmark for design rtl generation with large language model". In: *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2024, pp. 722–727.

[33] Deepak Vungarala et al. "SPICEPilot: Navigating SPICE Code Generation and Simulation with AI Guidance". In: *arXiv preprint arXiv:2410.20553* (2024).

[34] Hengrui Zhang et al. "A hardware evaluation framework for large language model inference". In: *arXiv preprint arXiv:2312.03134* (2023).

[35] Synopsys. *Synopsys VC Formal*. 2025. URL: https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html.

[36] Synopsys. *Synopsys SpyGlass*. 2025. URL: https://www.synopsys.com/verification/static-and-formal-verification/spyglass.html.

[37] Synopsys. *VCS T-Prop: Propagating Taint to Find Security Weaknesses*. Whitepaper. 2025. URL: https://www.synopsys.com/verification/resources/whitepapers/vcs-t-prop-wp.html.